hyper.dev

ⴰⵎⵓⵔⵓⵛ ⴰⵎⴰⵣⵉⵖ ⴱⵓⴱⴻⴽⴽⵉ

Amirouche Amazigh BOUBEKKI

- hyper.dev
- dx - 0 - a work in progress
- mezangelle - 2 - an open letter to mez breeze
- seed - 0 - codex: dubito, ergo cogito, ergo sum
- seed - 1 - chaos: the last question
- dx - 1 - code crafting
- babelia - 1 - moteurs de recherches precis ou flous
- letloop - 1 - micro-benchmarks do not tell the whole story
- möbius - 1 - only an open civilization is up to the task
- möbius - 2 - exwen
- floss - 1 - L'enclosure du code: avant les plateformes
- floss - 2 - L'enclosure du code: l'ère de l'assemblage
- floss - 3 - L'enclosure du code: l'ère générative
- floss - 4 - L'enclosure du code: le prélude et l'après
- möbius - 3.01 - the invariant - the hold
- möbius - 3.02 - the invariant - the garage lab hacker
- möbius - 3.03 - the invariant - the polyglot programmer
- möbius - 3.04 - the invariant - the burned-out senior dev
- möbius - 3.05 - the invariant - the kid in tizi ouzou
- möbius - 3.06 - the invariant - the pl theorist
- möbius - 3.07 - the invariant - the coordinator
- möbius - 3.08 - the invariant - the civilizational thinker
- möbius - 3.09 - the invariant - the air cycle
- möbius - 3.10 - the invariant - the check
- möbius - 3.11 - the invariant - departure
- möbius - 3.12 - the invariant - the privateer
- möbius - 3.13 - the invariant - the citizen
- möbius - 3.14 - the invariant - the educator
- möbius - 3.15 - the invariant - the connector
- möbius - 3.16 - the invariant - the symbiotic ai
- möbius - 3.17 - the invariant - the anchor backlog

- möbius - 3.18 - the invariant - the refactor cascade
- möbius - 3.19 - the invariant - the memory wipe
- möbius - 3.20 - the invariant - the artist
- möbius - 3.21 - the invariant - the silence
- möbius - 3.22 - the invariant - the person who was stolen from
- möbius - 3.23 - the invariant - the independent researcher
- möbius - 3.24 - the invariant - the community steward
- möbius - 3.25 - the invariant - the academic researcher
- möbius - 3.26 - the invariant - the representative
- möbius - 3.27 - the invariant - the manifesto
- möbius - 3.28 - the invariant - the invariant
- möbius - 3.29 - the invariant - the language born in transit
- möbius - 3.30 - the invariant - the first commit
- möbius - 5 - a content-addressed Scheme that speaks your language
- möbius - 6 - Product Brief
- möbius - 0 - Toward An Ecology Of Memory and Computation
- dx - 2 - The Ladder Nobody Built
- möbius - 7 - From Crisis To Commons
- möbius - 8 - Une civilisation
- möbius - 9 - what the hash cannot prove
- dx - 3 - Maîtriser nos outils passe par maîtriser la recherche
- dx - 4 - Reading mutation.py: a story
- dx - 5 - code crafting codex
- dx - 6 - public functions cognitive load
- dx - 7 - How to choose a database?
- mezangelle - 0 - defragmentation
- mezangelle - 1 - Do It Yourself: an artificial intelligence

- purview - 45 - 2026, week 45
- dx - 8 - AI without IA
- dx - 9 - The story keeps repeating
- möbius - 10 - Je vous apprends la langue du feu
- purview - 46 - 2026, week 46
- purview - 47 - 2026, week 47
- purview - 48 - 2026, week 48
- möbius - 11 - bb.py v0.6.0, qui a écrit ça, et qui l'a lu
- Colophon

hyper.dev

Azul, Salam, Namaste, Ni Hao, Hola, Salut, Hello,
I am ⍙⊑⋉⍜⁝⍁⍦ born ⁝⍧⍱⋉⊙ .
I love to code, and share knowledge.

May they / she / he beware;
today feels urgent as always
but only because we hope and work for a tomorrow.

I work for diversity, I work for peace, I work for justice, I work for progress.

Editorial — Pourquoi je rouvre hyper.dev

2026 / 03 / 15

Today feels urgent as always but only because we hope and work for a tomorrow.

I work for diversity, I work for peace, I work for justice, I work for progress.

This space is not for me. It is for my children, and if time allows, for their children. So that when they look back they can recognize something precise: that despite systemic bias, despite hardship, despite the noise — I remained relevant. Not famous. Not funded. But present, and pointing in the right direction.

Technologically relevant is a specific claim and I want to be specific about it. Not relevant because I followed the current. Relevant because something older than the current guided the

choices. A trans-generational intuition, the kind that comes from people who learned survival before they learned comfort, combined with what I believe was sound rationalism: the instinct to bet early on Scheme, on ordered key-value stores, on the comprehensible over the convenient. Those bets were right. I wish they had landed with more impact — on the craft, and on my life. That is an honest thing to say.

I did not arrive at this alone. Family held the space. The communities I passed through — some still alive, some dispersed — taught me that the work was possible before I could prove it. That is not a small thing.

Vision matters — this bears repeating. Not as comfort. As precision instrument. Vision is what makes refusal legible. Without it refusal is just stubbornness.

None of this is finished. It will never be finished. That is not failure. That is the nature of the work.

If success comes it will be a short moment — between the bitterness that precedes it and the forgetting that follows. Notice that moment. Enjoy it fully. Then return to work.

dx - 0 - a work in progress

2018 / 01 / 01 · https://hyper.dev/2018/dx-0-a-work-in-progress/

There is a painful thought I fight a lot when I publish something on the Internet: is this shameful? Will people see it as pretentious? I am sure it won't stop today, nor tomorrow.

I have dozens of repositories online, and most of them don't work anymore. I feel some shame about it; I have the feeling it gives a bad impression. Still, my belief that they will be useful, to me and to other people, is stronger.

I keep telling myself that a) I do the right thing by sharing my experience, b) it is good to talk about it. It is in this spirit that I blog, share code, and do the other things of life.

I have the right to write a software trying to understand how OpenCog works, and call it culturIA. To think in terms of Intelligence Augmentation instead of Artificial Intelligence, after Douglas Engelbart's Augmenting Human Intellect (1962). To use that very same name for a versioned hypergraph, and then mix it with my grandiloquent blog engine (read: a blog that is not static html) that I call a personal knowledge base. I should feel free to put Natural Language Processing down because it is too hard for me today, and to pick it up again. I should feel free to say cyberspace transderivational search without feeling ashamed. I should feel free to reinvent the wheel (on my free time (at least)). To create a search engine, fail, and try again later. To think that I can do better than others, and to be shown wrong. I should feel free to be a creative mind.

I have the right to dream big.

None of this is shameful, and dreaming out loud is not pretension. Still, I don't feel that way.

There should be a place for everyone, I hope one day I will be comfortable at mine.

mezangelle - 2 - an open letter to mez breeze

2021 / 02 / 21 · https://hyper.dev/blog/dear-mez-breeze/

Dear Mez Breeze,

It has been long time. I turned 30-something last week. This year also the 10th year I am working professionally as software engineer. Ten years ago, is also the time when I discovered your work. I remember in 2011, I read “Program or be programmed” by Douglas Rushkoff, and “The Filter Bubble” by Eli Pariser. Obama was president of the USA. I was trying to translate “Culture and Empire” by Pieter Hintjens in french. My domain name was something starting with hypermove, far fetched reference to Oxmo Puccino song “On ne danse pas”:

That was validated by Ox himself. It was the beginning of Google+. I wanted to build a social network, reminiscent of my most popular previous project on top of Neo4J(ava), with the help of Tinkerpop, Gremlin, and my biggest time sink, and most dangerous programming language: Python.

I wanted an hypercube as the logo for wanna-be my hypersocial website. At the time, a socialite was defined by Google search engine as the one that likes to share. That is how I stumbled upon Augmentology blog. I remember that, not because I have giant memory. I remember all of that because it all makes sense in my life. You, explained a lot of the things I had doubt about. You were the concrete realization of many of my dreams, and questioning answers commonly accepted as truth. Even if you are noticeable and noticed, you take positions that go unnoticed or silenced around me. It is or least part of my education, partly influenced by various aspects of

my life that getting noticed is not a good thing. After all, my dream is a continuous mesh of person with equals rights and opportunities where any of them can enjoy freedom, fulfill their passion, and realize their dreams. You have a Wikipedia page. That is where I discovered about Mezangelle. Mezangelle made my mind, and my skills stronger. A Big Sister. Eventually there was hope. I crafted some lines myself, even got encouragement by you and others.

I kept a sugared version of my favorite line:

```
# git commit --message="mezangelle: revert silence for a peaceful sanity"
```

I unleashed Mezangelle full steam hidden in a very common feature such as an alias: amz3. The official explanation is the following:

> amz3 is a compressed representation of a morpho-syntactic combinatorics transderivationnal search in a graphical semantic space with the help of pattern matching that aggregates several senses. The induced experience while reading it is a reference to the Isaac Asimov novel «Little Lost Robot» in practice an obscure form of poetry named Mezangelle.

Karine and you were the ones that gave me the strength, courage and desire to voice myself.

What remains of that stream of thought is one question: “Should I learn French?”. A couple of month later, you requested a translation of one piece I wrote and recorded. I did my best, and you told me that it was great. It is not about you or for you. It is time to break the spell. It is for nobody, and

everybody in particular. I was meant to taunt. It is not perfect, I like it, even with the imperfections. There is a particular imperfection, that I do not want to take away, I was merely trying to voice part of the infinite and meticulous poetry that makes up Mezangelle. Any time I think about it, I remember about you, Mez Breeze and the perfect deep drum of senses rhythm that are spelled out in your Mezangelle. Your question about learning french was impossible to answer. “Choose you own destiny”. And the ever lasting question: For me? In fact, the last verses are a tribute to Mezangelle. And even the last line can only start to make sense if / when you both know french and Mezangelle.

It is difficult today to tell what exists thanks to you, and what is my own deed.

You may or may not have invented Mezangelle, maybe all Mezangelle is a invention by you. My mother tongue and its associated history is a lie. Still, you are the one that wield it when I needed it.

eng4g3!

seed - 0 - codex: dubito, ergo cogito, ergo sum

2021 / 06 / 04 · https://hyper.dev/blog/codex-dubito-ergo-cogito-ergo-sum/

Last couple of weeks I learned about:

- 3-LISP: a theorical evolution of LISP where everything can be inspected;
- Kernel: an evolution of Scheme that forgo syntax transformers as described in Scheme;
- Control Flow Analysis / Just-In-Time and Ahead-Of-Time compilation / optimizations: what I call “dubito”. It is known as partial / symbolic / algebraic evaluation, and includes type inference.

My goal is to devise and build an optimizing compiler which does not precludes AOT or JIT. In other words: I want both AOT and JIT. As a subgoal, I want to build an optimizing or optimized pattern matching program in the spirit of SRFI-200, and SRFI-204, that I call pattern combinators. One particular optimization, I think about is what Software Design for Flexibility [SDF] calls unification.

For instance given the following `match` expression:

```
(match exp
  ((a) out1)
  ((a d) out2))
```

Can be rewritten as follow:

```
(match exp
  ((a . rest) (match rest
```

```
(() out1)
((d) out2))))
```

In that case, doing some kind of prefix compression.

`match` can be seen as a pattern combinator, that is a parser combinator that can traverse not only a sequence, but also a sequence made of nested sequence in a recursive way. For the purpose of the pattern combinator, a sequence is a finite acyclic ordered set. A sequence might be a list, vector or an ordered mapping. A parser combinator such as the one described in [GLL] is a good basis for a pattern combinator, except if does not support the optimizing / optimized part.

To be able to implement optimizing pattern combiner, it would require to reflect upon the match predicates, in particular have access to a predicate `subsume?` that allows implement a partial order over predicates. And that should be done at in the object language.

The object language is the language of the user. It is defined in opposition to meta language that is used to implement the object language. The meta language might be itself implemented in terms of another meta language until… until all the way down! That is where 3-LISP call into atoms, the physical atoms (not the atoms of the simulation).

We will just glimpse over the problem(s), and add a reflection applicative called `sum`, so that we can introspect how applicative and operative are made, so that in turn we may be able to optimize code at runtime from the object language.

Instead of trying to implement `subsume?` and the optimizing pattern compiler, I will implement only arithmetic folding (replace an arithmetic computation with its value) with an

applicative based on the inferences / knowledge / theories built by an applicative called `dubito`.

So, given the following pseudo-code:

```
(define (make-frob-adder value)
  (lambda (other)
    (+ other value 36)))

(define my-frob-adder-2 (make-frob-adder 2))

(my-frob-adder-2 4) ;; => 42
```

A simple compiler will just compile `my-frob-adder-2` into a `lambda` that is passed an extra environment argument called `static` so that the resulting procedure does not have free-variables, it will look like:

```
(define (my-frob-adder-2-compiled static other)
  (define value (environment-ref static 'value))
  (+ other value 36)
```

The trick is that if we forgo the last line of the previous program, the closure of `my-frob-adder-2` is known at compile time. If we also consider the last line, the whole program can be compiled to `42`: That is what is called (AFAIK) constant folding.

Let's consider the following program:

```
(define (make-frob-adder value)
  (lambda (other)
    (+ other value 36)))

(define my-frob-adder-2 (make-frob-adder 2))

(my-frob-adder-2 (string->number (cadr (command-line))))
```

It is similar to the original program, except it `my-frob-adder-2` takes its argument from command-line, hence it can not be folded into an integer object, because… `(command-line)` is not known at compile time. So, indeed after a simple compilation, we end up with the following code:

```
(define (my-frob-adder-2 static other)
  (define value (environment-ref static 'value))
  (+ other value 36))

(my-frob-adder-2 (alist->environment `((value . ,2))) (string->number (cadr (command-
          line))))
```

That is naive example: an optimizing compiler can create the following optimized definition for `my-frob-adder-2`:

```
(define (my-frob-adder-2 other)
  (+ other 40)) ;; 36 + 4 = 40
```

So what is this all about?!1!

It is also seems a little naive given that Chez Scheme can compile to native code an s-expr at runtime, given a perfomance timer (cost center), and counters, that gather new knowledge only known at runtime, it is completly possible to implement a manual JIT [MANUAL-JIT].

So what do `sum` do that is new? Not much: it puts all that together.

Kernel [R-1RK] will factorize macro and procedure into operatives, it will also reify all environments: static / lexical, and dynamic. The application `sum` will reify in the object language the implementation of any object… including `eval` in other words, `eval` can be reifyed as a meta-evaluator or better as the “source code” in the previous meta language. `sum` does not only mean that the source is embedded in the program, but

also prescribe that it is possible to represent in a possibly infinitly resursive way all the meta-languages. The limit being our knowledge, time, and energy to encode that as source. Does it mean that this language is self-bootstrappable, in other words that it does not require a seed: no. Even it it self-hosted and self-describing, it requires a seed [BOOTSTRAPPABLE].

In the above paragraph I glossed over `(sum eval)`: i) does it return a meta-evaluator, a program represented in the object language that can evaluate the object language (itself), ii) does it return an evaluator in the previous meta-language. Both are possible, the latter is more interesting, because it expose the compiler tower [NANOPASS] to the object language.

`eval` is not the only interesting primitive procedure. In the context of Scheme, it is interesting to reflect upon a couple of procedures. One of the most interesting is `call/cc`, another interesting procedure is the non-standard `expand` and yet another everything related to the garbage collector. To stay goal-driven, a target application of that reflection, that I think may be interesting, can be to disable or customize the garbage collector for a subset of the program. Another goal might to change for a subset of the program the evaluation strategy without relying on meta-evaluation [EVALS].

In the case where `(sum eval)` returns the source of `eval` in the previous meta-language, `expand`, `call/cc`, and the garbage collector should be represented… somehow!

References

- [SDF] https://mitpress.mit.edu/books/software-design-flexibility
- [GLL] https://epsil.github.io/gll/

- [MANUAL-JIT] https://m00natic.github.io/lisp/manual-jit.html
- [R-1RK] https://web.cs.wpi.edu/~jshutt/kernel.html
- [BOOTSTRAPPABLE] http://bootstrappable.org/
- [NANOPASS] https://nanopass.org/
- [EVALS] https://en.wikipedia.org/wiki/Evaluation_strategy

seed - 1 - chaos: the last question

2021 / 06 / 05 · https://hyper.dev/blog/chaos-the-last-question/

I am investigating this pattern combinator thing, because pattern are everywhere in programming e.g Chomsky's grammars, and also Machine Learning. The thing I am trying to build is different from Machine Learning pattern recognition. Pattern combinators are neither supervised nor unsupervised, they do not learn, they are told with a specification constructed with a combination of procedures how to recognize patterns.

Pattern matching facility ease the work of constructing an interpreter or compiler. They are the basis of Scheme syntax-rule, syntax-case, nanopass framework, Kernel, and GNU Guile compiler tower [GNU-GUILE]. Part of Guile compiler tower, there is `foldt*` by Andy Wingo. `foldt*` builds upon Kiselyov's `foldt`. Both `foldt` and `foldt*` allow to recognize patterns in s-expr, they also allow to construct a new object (just like regular `fold`). Pattern combinators are concerned about traversing the source; the patterns in that source are recognized with user provided procedures (fhere, fdown, fup); pattern combinators always contruct an object following the same template, that is a flat environment, such as a mapping or association list.

In the previous note, I have glossed over `sum` that re-surface the implementation of a procedure. `dubito` some kind of inference engine. `cogito` was barely mentioned, but I meant it as an `eval`.

In this note, I will try to explain how `(sum cogito)` works.

Here is the *Shutt's Equation of Software*:

```
(define eval
  (lambda (exp env)
    (cond ((kernel-pair? exp)
           (combine (eval (kernel-car exp) env)
                    (kernel-cdr exp)
                    env))
          ((symbol? exp)
           (lookup exp env context))
          (else exp))))
```

This is a slightly modified version of `SINK`'s `eval`, I dropped the `context` that is the reification of a continuation, that in `(call/cc proc)` is passed to `PROC`. Unlike in Scheme, a context is encapsulated ie. it is its own type (see Racket and Gambit).

Tentative emulation of Kernel with Scheme

About Scheme, I will try to describe the semantic of Kernel using Scheme without actually building an interpreter in Scheme (that is what SINK does). Instead, I will describe a subset of Kernel in particular, the `$vau` operative, that allows to create new operative with Scheme pseudo-code:

```
(define-syntax $vau
  (lambda (stx)
    (syntax-case stx ()
      (($vau args env body ...)
       #'(lambda (args env) body ...)))))
```

That is wild guess ie. preliminary translation. To call a procedure constructed with `$vau` you need another syntax transformer and a few helpers:

```
(define-syntax apply-vau
  (lambda (stx)
    (syntax-case stx ()
```

```
((apply-vau operative args ...)
 #'(let ((dynamic-environment (environment-current)))
     (operative 'args dynamic-environment))))))
```

(Now that I think about it, it can be implemented in terms of `syntax-rule`)

Mind the quoted `args`, `args` is constructed in the template as a quoted expression, and its evaluation might be done in `body ...` with the help of Scheme's `(eval exp environment)`.

`environment-current` is like Guile's `the-environment`.

Another emulation of Kernel with Scheme

Another approach is to use only procedures and no macros. For instance:

```
(define (my-operative args env)
  body ...)
```

Since there is two kinds of callables in `Kernel`: operative and applicative. One will need to prefix every Scheme call with a `kernel-call`:

```
(define (kernel-call combiner env args)
  (if (applicative? combiner)
      ((unwrap combiner) (map (lambda (arg) (eval arg env))))
      ;; otherwise, it is an operative
      (combiner args env)))
```

Where `applicative?` is the predicate associated with:

```
(define-record-type <applicative>
  (wrap operative)
  applicative?
  (operative unwrap))
```

Then what you write in Kernel as:

```
(combiner a b c)
```

Would be transpiled as:

```
(kernel-call my-combiner (environment-current) (list 'a 'b 'c))
```

`(environment-current)` current can be constructed at compile time, since it is the static / lexical scope (I hope).

The only missing piece I can forsee is that Kernel support trees as operands, but that requires more work, including a pattern matcher!

`sum`, or no `sum`

Previously, I wrote the goal of `sum` is ultimately to call `(sum eval)` and return an object language representation of `eval` in the meta-language. In other words, it would return the full source of the compiler, one language-level below the object language.

I will investigate `(sum +)`: what is the meta-language definition of the applicative `+` in Scheme:

```
(define (sum +)
  (applicative (operative (lambda (args env) (+ (env-ref env 'a) (env-ref env 'b))))))
```

It does not make sense to compile a mere `lambda`, you need a full lambda such as:

```
(lambda (args env) (+ (env-ref env 'a) (env-ref env 'b)))
```

It would be transpiled to the following web assembly:

```
(func $a.683 (param $cl.397 externref) (result i32) (result externref)
      (local $a externref)
      (local $b externref)
      (local $out externref)
      (local.set $k (table.get $stack (i32.const 0)))
```

```
(local.set $a (table.get $stack (i32.const 1)))
(local.set $b (table.get $stack (i32.const 2)))
(local.set $out (call $+ (local.get $a) (local.get $a)))
(table.set $stack (i32.const 0) (local.get $out))
(i32.const 1) ;; continue = yes
(local.get $k)) ;; continuation closure
```

The previous text would be the representation of `(sum (sum +))` in the object language. More or less.

References

- [GNU-GUILE] https://www.gnu.org/software/guile/docs/docs-2.2/guile-ref/Compiler-Tower.html

dx - 1 - code crafting

2023 / 09 / 01 · https://hyper.dev/blog/dx-1-code-crafting/

A problem can have several solutions. A solution can have different implementations each of which with their own strengths and weakness in terms of space, time and generality.

The goal of a code experience design is to optimize the cognitive effort required to understand, learn and use. Otherwise said, code usability aim to improve ease.

While looking for ease, one might trade it over expressive power.

Crafting an experience requires to understand the cognitive load associated with the production. Cognitive load theory has been designed to provide guidelines intended to assist in the presentation of information in a manner that encourages learner activities that optimize intellectual performance.

Here are some of those guidelines:

- People learn more effectively when they can build on what they already understand;
- Take advantage of schemata;
- Take advantage of visual thinking and other non-verbal thought;
- The average person can retain ~~only seven~~ five chunks ~~of information~~ in short-term memory;

Repeat!

babelia - 1 - moteurs de recherches precis ou flous

2024 · https://hyper.dev/2024/moteurs-de-recherche-precis-ou-flou/

un moteur de recherche est précis

Cet outil est construit pour enquêter dans des bases de données structurées. Les données structurées sont organisées en documents de plusieurs parties aussi souvent appelées champs, et où chaque champ a une typologie stricte, qui ne varient pas de document en document : cela peut être une date de publication, un numéro de page donc un nombre entier, un code de bibliothèque, les champs peuvent être même une chaine de mot assembler d'espace et de ponctuations : un texte. Lorsque le document a un champ corps, fréquemment du texte, l'habitude est d'appeler les autres champs, les métadonnées.

La cible d'un moteur de recherche précis est surtout des données déjà organisées ou organisées automatiquement ou semi-automatiquement. L'avantage d'une structuration systématique est de faciliter des traitements. Les traitements visent à mettre en perspective la base de données, en regardant les données de face, mais aussi de profil, au rayon X, etc. C'est possible en faisant une ou plusieurs étapes, des calculs et des transformations, en projetant, en rassemblant, en agrégeant les champs disponibles pour en créer de nouveaux. L'objectif est de favoriser la création de vues variées autour de la demande d'aide de l'utilisateur. En d'autres termes, cela permet une aventure multidimensionnelle, un tableau avec plusieurs colonnes. La matrice sous-jacente se retrouve dans la représentation à travers les filtres de différents types qui sont

agrémentés de compteurs. Cela s'appelle des facettes, celles-ci rendent accessibles des informations préexistantes, mais auparavant difficiles, voire impossibles à appréhender sans traitement informatique.

Ces moteurs de recherche sont couteux en mise en place et en maintenance, car il faut déterminer à plus ou moins long terme comment se feront les calculs et lesquels sont personnalisables par l'utilisateur. Aussi, il faut savoir combien de temps prennent ces calculs. S'ils prennent trop longtemps, il faut prévoir de précalculer ceux-ci en amont et de les figer, ce qui nécessite du stockage durable. Dans un moteur de recherche précis, la demande de l'utilisateur est précise et le moteur de recherche doit s'y conformer dans les détails pour ne pas fausser la réflexion. Cela étant, le moteur de recherche précis peut agrémenter la liste de résultats d'éléments connexes issus du flot de résultats qui suggerent d'autre pistes.

un moteur de recherche est flou

Ce type d'appareil se concentre sur la fouille de texte. Le texte peut être accompagné d'une table des matières. Celui-ci est opéré en grande partie à l'aide d'un langage informatique construit autour de la langue naturelle, avec des mécanismes comme les opérateurs booléens, les « phrases à rechercher », et qui peuvent être triés par la date de découverte. La matière première de ces logiciels sont des documents peu ou pas structurés. Le flot de résultats est toujours présenté sous forme de liste avec un titre, un extrait ou un résumé et une référence durable comme une adresse web.

Le moteur de recherche flou préfère les documents de taille modeste autour de mille mots. Ces moteurs de recherche fonctionne aussi sur des documents de la taille d'un

paragraphe autour de 50 à 100 mots. Les grands formats peuvent être pris en compte en découpant ceux-ci en plusieurs segments.

Ces moteurs de recherche ne sont pas stricts sur l'interprétation des demandes de l'utilisateur. En effet, dans le cadre de recherche floue, le souhait de l'utilisateur est de faire une recherche par sens ou autour de sa demande qui, dans les premiers pas d'un travail de recherche, est habituellement imprécise. En d'autres termes, la demande est un indice que le moteur de recherche suit pour aiguiller l'utilisateur dans sa péripétie intellectuelle.

letloop - 1 - micro-benchmarks do not tell the whole story

2026 / 03 / 29 · https://hyper.dev/2026/chez-scheme-letloop-transparent-async-microbenchmark/

I published a benchmark report comparing six HTTP server implementations across Rust, Scheme, JavaScript, and Python. Scheme (flow2 + io_uring) placed second at 306k req/s — behind Rust (axum) at 407k, level with Bun canary (303k) and Bun (294k), and ahead of Node.js and FastAPI.

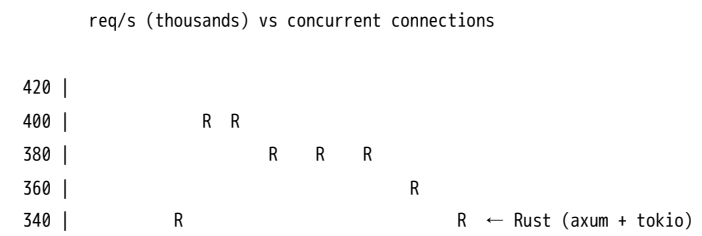
The Scheme server now runs on flow2, letloop's concurrency library, and on nothing beneath it: every read, write, timer, and accept goes through flow2's public API.

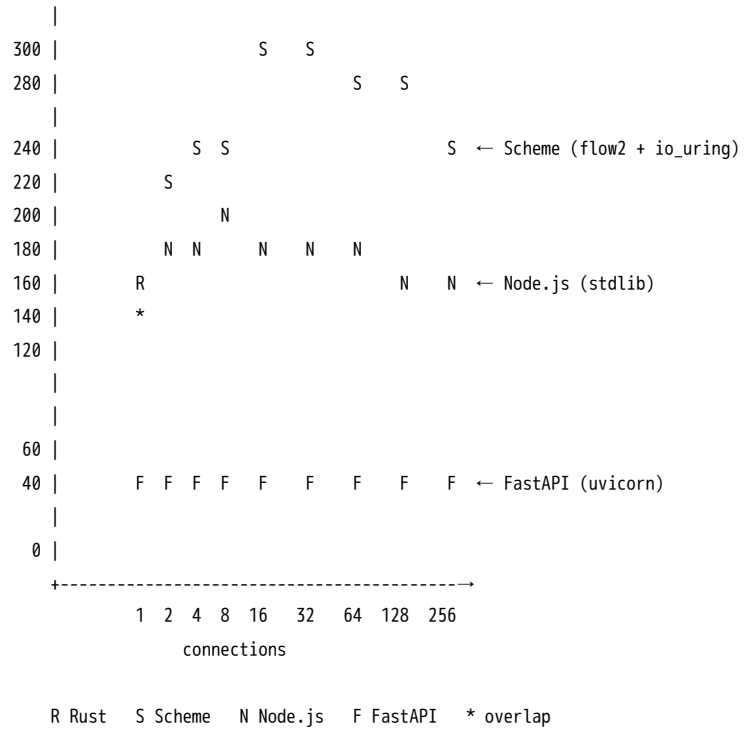
These are micro-benchmarks. I want to be honest about what they show and what they don't.

what the numbers say

A trivial counter endpoint. Single-threaded. Ten seconds per concurrency level. `wrk` as the load generator. No disk, no database, no parsing, no serialization. The purest measure of "how fast can this runtime shuffle bytes through a socket."

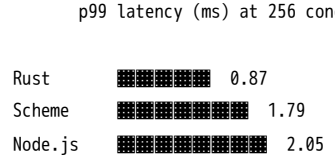
Here's how throughput scales with concurrent connections for four runtimes that represent four different async strategies:

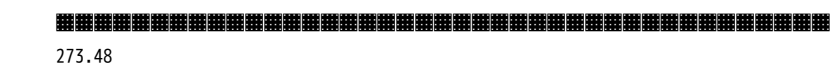




Rust peaks early at 407k req/s (8 connections), then declines gently to 347k at 256 — the hallmark of a mature runtime with a well-tuned scheduler. Scheme peaks later at 306k req/s (16 connections), holds near 300k up to 64, then eases to 234k at 256. Node.js sits flat around 190k from 4 connections onward, sagging to 160k at 256. FastAPI never breaks 37k — the Python GIL is the ceiling, not uvicorn.

But the more interesting picture is latency under pressure:





At 256 connections, Scheme's p99 is 1.79ms — about twice Rust's 0.87ms, and a little below Node.js's 2.05ms. FastAPI's p99 explodes to 273ms — a hundred and fifty times worse than Scheme. This is where architecture starts to show through the numbers.

what the numbers don't say

Nothing about correctness. Nothing about how the code reads, how it changes, how it fails. Nothing about what happens when you add a database, a parser, a state machine, an authentication layer. Nothing about the six months after the benchmark, when the system has to evolve under production pressure and the original author is gone.

A micro-benchmark is a controlled experiment. It holds everything constant except the variable you're measuring. That's its virtue and its limit. The variable here is raw I/O throughput. Everything that makes a real system hard — concurrency bugs, resource leaks, error propagation, compositional complexity — is absent by design.

why it still matters

The benchmark exists to answer one narrow question: is the letloop async runtime fast enough that choosing Scheme doesn't mean accepting a performance tax? The answer is yes. At 306k req/s with a p99 under 1.8ms at high concurrency, flow2 + io_uring is level with Bun, behind Rust, ahead of Node.js, and dramatically ahead of Python.

That matters because the choice was never about speed. It was about something else.

transparent async

Most async runtimes are colored. In Rust, you write `async fn` and enter a different world — different traits, different lifetimes, different debugging. In JavaScript, you write `async/await` and functions split into two kinds: the ones that return promises and the ones that don't. In Python, the split is even worse — `asyncio` is essentially a parallel universe that doesn't compose cleanly with synchronous code.

letloop's async is transparent. You write ordinary Scheme. The runtime handles scheduling via io_uring underneath. There is no function coloring, no `await` keyword, no split between "async code" and "normal code." A procedure that reads from a socket and a procedure that computes a factorial use the same call conventions, the same continuation model, the same debugging tools.

This is not a cosmetic difference. Function coloring has real costs:

It doubles the API surface — every library that does I/O needs sync and async variants, or forces callers into one world.

It makes composition harder — mixing sync and async code is where most concurrency bugs live in Python and JavaScript.

It raises the floor of understanding — a new developer can't ignore the async/sync distinction; it's load-bearing from day one.

Transparent async removes these costs. The entire system has one calling convention. Composition is ordinary function

composition. The floor of understanding is lower. Not zero — io_uring itself is complex machinery — but the complexity stays below the API boundary where it belongs, instead of leaking into every function signature.

what's next

chez scheme + pico http parser + io_uring is a work in progress. The scheduler is single-threaded. The error handling story is incomplete. These are real gaps, not aspirational roadmap items.

But the foundation is sound. The numbers confirm that the architecture doesn't impose a performance ceiling. What matters now is everything the benchmark doesn't measure: correctness, composability, comprehensibility. The things that make a system livable for the person who maintains it after the benchmarks are forgotten.

The full report with all six implementations is at hyper.dev/letloop/report-iii.html.

möbius - 1 - only an open civilization is up to the task

2026 / 03 / 31 · https://hyper.dev/2026/open-civilization/

Dear Janko,

I think I know the turmoil you are describing. Asymmetry is not new. For billions, it has always been the norm. There is no outside.

But there is between.

There are tools within to instrument the forest, teach the villagers to build lanterns, and let the ideas we exchange swarm around those artificial lucioles for a while.

Once Rushkoff said: program or be programmed. I took it — not as rage against the machine, but as an opening — more degrees of freedom: a neolove letter, an interactive mashup at a party, DJs mixing the same party across cities, abroad, and across oceans 🌊.

Möbius started small: attributing a micro-contribution, a single function, through a linked network of git repositories. From there you can imagine that with some of the power of our weighted anti-hero it becomes a coordination layer — a way to not write the same thing twice, to find nearby peers. The multilingual cognitive diversity is almost an accident of the design. Almost. Seeding the dark forest with diverse multilingual cognitive patterns — a silver lining?

After a nap I sat behind the keyboard. It is soon my brother-in-law's anniversary — a good opportunity to use the pseudo-random animation I published to Möbius a couple of weeks

ago. I was made aware that someone in Alaska used it at a local LAN party, lighting up the event based on the mood captured by a phone's mic. I am happy and thankful. My project will notify back like a boomerang through the Möbius network that new parties were enlightened by that contribution, by way of a new derivation. When derivation is structural — hash pointers, not mutable registries — absorption stops being silent.

Call it fiction for now — the kind that becomes infrastructure once enough of us act as if it were true. And I have been dreaming of it for a decade — whether LLMs party within the enlightened dark forest is irrelevant for the time being.

Whether the attribution mesh gets proper recognition is in our hands. We build it each day, or we don't. Making each day less boring, more joyful, less painful, and more open to possibilities.

Möbius is a challenging open channel. Only an open civilization is up to the task.

Möbius series: - Toward An Ecology Of Memory and Computation — the vision - Exwen: property-centric data management — layer 2 - bb.scm: content-addressed Scheme — implementation - Product Brief — technical specification - From Crisis To Commons — the narrative - What The Hash Cannot Prove — security considerations - Une civilisation équilibrée… — philosophy

möbius - 2 - exwen

2026 / 04 / 08 · https://hyper.dev/2026/mobius-exwen/

Why this exists

Imagine a student in Tizi Ouzou builds a small database of Kabyle proverbs. Each proverb has a text, a translation, a theme, maybe a region where it's commonly heard. She doesn't study database design. She just describes what she knows: this proverb has these properties.

Now imagine a researcher in Osaka is cataloguing yōkai folklore. Different language, different domain, same gesture: objects with properties.

Neither of them should need to design a schema, choose a table layout, learn SQL, or ask permission from a specification committee. They describe what they have. The properties speak for themselves.

Exwen is a proposal for property-centric data management.

What's wrong with tables

SQL organizes data around tables — rigid containers that decide in advance what shape your data must take. An object exists *because* it belongs to a table. Its properties are columns, fixed at creation time.

This gets the practice backwards. We come to understanding through sens first, then words, then properties, then eventually objects.

```
s(ens) → wor[l]ds → proper[ties] → obj[ect⋯]s
```

We don't start with the container and fill it — we start with meaning and let form follow. A proverb is a proverb because it carries wisdom and has a text, not because someone created a `proverbs` table first. A person is not a row in `users`.

The table-centric model forces you to know the shape of your data before you understand it. Exwen starts where you start: with meaning.

Properties

In exwen, objects exist. They have properties. Each property has a definition — possibly multiple definitions across different contexts and communities.

An object is not typed by its table. An object *becomes* comprehensible through the properties it carries. If something has `text`, `translation`, and `region`, you can work with it — whether you call it a proverb, a lexical entry, or something else entirely.

Systems discover what objects are by examining what properties they have. When an object matches a predicate — a pattern of properties — it becomes usable in a given context.

Others have explored parts of this territory. Exwen draws on that work and proposes a coherent foundation: the property is the primary unit of data modelling.

No schema police

Properties can have multiple definitions. In a well-established domain, communities may converge on consistent usage — not because a standards body enforced it, but because good definitions are comprehensible.

But convergence is not the goal. A geocities page built with `<center><bold>` told stories too. Algospeak, mezangelle, improvised markup — these are not anti-patterns. They are how people actually express themselves when given a surface to work with.

Exwen does not punish divergence. What it does is keep things connected. However you name your properties, however strange your structure, your data is still reachable, still queryable, still part of the graph. The well-named properties are easier to find. The eccentric ones still tell their story.

Who reads properties

Humans, obviously. But also: language models, agents, scripts, browsers.

A property-centric store is inherently more accessible to diverse clients than a table-centric one. An LLM doesn't need to reverse-engineer a schema. It asks: what properties does this object have? And works from there. A lightweight browser — one that doesn't run arbitrary JavaScript but instead queries structured data — can navigate property-centric stores directly.

This is why exwen matters beyond database theory. It's the layer that turns opaque storage into legible, queryable, navigable data — for anyone, in any language, with any tool.

Where exwen sits

Exwen is not a database. It is a view on data — a way of organizing access to what's stored underneath.

Below exwen sits an ordered key-value store (okvs): silent, opaque, mathematically grounded, indifferent to the shape of what it holds.

Above exwen sit the tools people use: functions that transform and query properties, browsers that navigate them, agents that reason over them.

Exwen is the middle layer that makes storage speak.

Who this is for

This is for anyone who has ever felt that their data didn't fit the boxes they were given. For the student in Tizi Ouzou, the researcher in Osaka, the developer in São Paulo who wants to describe what they know in terms that make sense to them — and have that description be enough.

Exwen is part of the Möbius project.

This letter was written in collaboration with Claude.

floss - 1 - L'enclosure du code: avant les plateformes

2026 / 04 / 19 · https://hyper.dev/2026/enclosure-du-code/1-avant-les-plateformes/

Comment le logiciel libre — né comme projet politique de liberté — a été recadré pour rendre l'extraction acceptable

Première session — La chaîne d'approvisionnement logicielle, des années 1990 à l'ère générative

← L'enclosure du code — Série complète

---

1. Synthèse

[INTERPRÉTATION] Le mouvement du logiciel libre est né dans les années 1980 comme un projet explicitement politique : défendre la liberté des utilisateurs face à l'appropriation privée du code. 🟢 90-100% En 1998, un groupe de technologues a délibérément remplacé le vocabulaire de la liberté par celui de l'efficacité commerciale, créant le terme « open source » pour rendre le mouvement acceptable aux yeux des entreprises. 🟢 90-100% Ce recadrage n'était pas une simple opération de marque. Il a constitué l'acte fondateur qui a rendu culturellement acceptable l'extraction de valeur à partir des communs numériques — tout en élargissant massivement la participation et les ressources disponibles. 🟡 70-89% La tension entre ces deux lectures — trahison d'un idéal ou compromis pragmatique nécessaire — n'est pas résolue. Elle structure aujourd'hui encore l'économie politique du logiciel.

---

2. Résultats détaillés

2.1 L'ère artisanale : le développeur et ses outils

[FAIT] Dans les années 1990, construire du logiciel signifiait maîtriser un environnement technique de bout en bout. Le développeur savait ce que faisait chaque composant de sa pile logicielle, souvent parce qu'il l'avait lui-même assemblé. Les gardiens de cet écosystème étaient trois empires : Microsoft contrôlait le bureau, Sun Microsystems possédait Java et les stations de travail, Oracle régnait sur les bases de données. Leur modèle économique reposait sur des licences propriétaires qui rendaient le développeur dépendant — et payant.

[INTERPRÉTATION] Cette dépendance était aussi une forme de discipline artisanale. Les développeurs de cette époque décrivaient fréquemment leur travail comme un métier (« craft »), où la compréhension profonde de ses outils était inséparable de la dignité professionnelle. Comme le note la sociologie des techniques en France, il existait un parallèle entre la « prolétarisation » décrite par Bernard Stiegler — l'ouvrier réduit à l'exécution par la machine industrielle — et la prolétarisation de l'utilisateur de logiciels propriétaires, condamné à ignorer le fonctionnement des programmes qu'il utilise. 🟡 70-89%

Le rapport entre le développeur et son outil était un rapport de propriété intellectuelle, dans les deux sens : l'éditeur possédait le code, mais le développeur qui maîtrisait pleinement son environnement possédait une forme de savoir irréductible. Cette époque a produit une identité professionnelle où « être développeur » signifiait comprendre, pas seulement utiliser.

2.2 Le logiciel libre comme projet politique

Les quatre libertés

[FAIT] Richard Stallman a lancé le projet GNU en 1983 et fondé la Free Software Foundation en 1985. Le mouvement repose sur quatre libertés : utiliser le programme pour n'importe quel usage, étudier son fonctionnement et l'adapter, redistribuer des copies, et améliorer le programme et publier les améliorations. 🟢 90-100% Stallman lui-même résume le logiciel libre en trois mots empruntés à la devise de la République française : « liberté, égalité, fraternité ».

[INTERPRÉTATION] Ceci n'était pas un argument d'efficacité. C'était un programme d'émancipation. Le logiciel libre était un mouvement qui cherchait à subvertir la propriété intellectuelle plutôt qu'à l'abolir — utiliser le droit d'auteur pour garantir l'ouverture plutôt que la fermeture. Comme le note le juriste Mikhaïl Xifaras, les licences libres fonctionnent précisément parce que le droit d'auteur reconnaît une appartenance première de la création à son créateur, ce qui permet d'« exclure la possibilité de l'exclusion ». 🟡 70-89%

Hors des États-Unis : une résonance politique distincte

[FAIT] France. La France est qualifiée par l'étude OpenForum Europe comme le leader des politiques consacrées aux logiciels libres pour le secteur public en Europe, avec un score de 65% devant l'Italie (63%) et l'Espagne (57%). La circulaire Ayrault de 2012 est le premier texte juridique français à reconnaître une égalité entre logiciel libre et propriétaire. La loi pour une République numérique de 2016 encourage l'usage des logiciels libres et des formats ouverts. L'ADULLACT travaille depuis 2002 à promouvoir le libre dans les

collectivités territoriales. Le marché de l'open source en France pèse désormais plus de 5 milliards d'euros. 🟢 90-100%

[FAIT] **Brésil.** Sous l'administration Lula, le gouvernement brésilien a promu l'adoption du logiciel libre dans tout le secteur public comme stratégie de développement et d'inclusion numérique. Le mouvement a émergé d'un réseau d'« experts insurgés » liés au Parti des Travailleurs et aux syndicats, notamment dans l'État du Rio Grande do Sul dès 1999. Lula a personnellement participé au FISL (Fórum Internacional Software Livre) à Porto Alegre en 2009, déclarant que le logiciel libre était « essentiel à la construction d'une société de l'information inclusive ». L'activisme pour le logiciel libre au Brésil s'est mêlé à l'activisme de gauche et a utilisé le pouvoir d'achat de l'État pour démocratiser l'accès au numérique. 🟢 90-100%

[FAIT] **Afrique francophone.** Des communautés actives du logiciel libre existent dans au moins 19 pays francophones, fédérées depuis 2021 par les Rencontres Amicales Francophones du Logiciel Libre (RAFLL), reliant Abidjan, Bamako, Cotonou, Dakar, Lomé, Niamey, Ouagadougou, Béjaïa, et d'autres villes. L'argument central est celui de la souveraineté numérique : comme le formule Wilfried N'Guessan, promoteur du libre en Côte d'Ivoire, adopter les logiciels libres revient à « défendre sa souveraineté numérique ». Myriam Criquet affirme que l'Afrique est une terre d'élection du libre parce que sa culture communautaire et son droit coutumier sont prédisposés à régir les droits et obligations par l'organisation des liens. 🟡 70-89%

[INTERPRÉTATION] Le logiciel libre a été reçu hors des États-Unis non pas principalement comme un modèle de développement logiciel, mais comme un levier de

souveraineté contre la dépendance technologique. En France, au Brésil, en Afrique francophone, le vocabulaire politique du libre — liberté, indépendance, bien commun — résonne avec des traditions politiques locales. C'est précisément ce vocabulaire que le recadrage « open source » de 1998 a délibérément abandonné.

2.3 Le recadrage stratégique : quand « open source » remplace « logiciel libre »

[FAIT] Le 3 février 1998, dans les bureaux de VA Research à Palo Alto, un groupe de technologues — dont Eric S. Raymond, Bruce Perens, Larry Augustin, Sam Ockman, Todd Anderson et, au téléphone, Jon « maddog » Hall — s'est réuni pour stratégiser après l'annonce par Netscape de la libération du code source de son navigateur. Le terme « open source » a été proposé par Christine Peterson, directrice exécutive du Foresight Institute, un think tank spécialisé en nanotechnologie. Richard Stallman n'a pas été invité à la réunion. L'Open Source Initiative (OSI) a été fondée fin février 1998. 🟢 90-100%

[FAIT] La décision était explicitement motivée par le désir de rendre le mouvement acceptable au monde des affaires. Comme l'écrit Peterson : l'ambiguïté du mot « free » (gratuit ou libre ?) détournait la conversation, et le langage de Stallman sur l'éthique et les « maux sociaux » aliénait les cadres d'entreprise. Raymond a synthétisé sa position dans un essai devenu célèbre, défendant le modèle ouvert sur la base de sa supériorité technique, non de principes moraux. 🟢 90-100%

[FAIT] Stallman a rejeté immédiatement le nouveau terme et refusé de rejoindre l'OSI. Sa formule est restée : « Open source is a development methodology; free software is a social movement. » Il a averti que la rhétorique de l'open source

détourne les utilisateurs des questions morales centrales, en brouillant la distinction entre logiciel libre et logiciel semi-libre ou propriétaire. 🟢 90-100%

[INTERPRÉTATION] Bruce Perens, cofondateur de l'OSI, a lui-même exprimé des doutes dès 1999, regrettant le « schisme » en cours. Il avait espéré que l'OSI servirait simplement d'introduction aux principes de la FSF pour les non-hackers, pas de mouvement rival. 🟡 70-89%

Le rôle d'O'Reilly Media

[FAIT] Tim O'Reilly a organisé en 1998 le « Freeware Summit » (rebaptisé Open Source Summit), invitant les leaders des principaux projets. O'Reilly est crédité d'avoir popularisé le terme « open source » auprès du monde des affaires, comme il populariserait plus tard « Web 2.0 » et d'autres termes structurant l'industrie. Le rôle d'O'Reilly Media n'était pas celui d'un simple éditeur : c'était celui d'un fabricant de récits — l'organisation qui définissait le vocabulaire dans lequel l'industrie pensait ses propres transformations. 🟡 70-89%

[INTERPRÉTATION] Ce rôle de mise en récit est crucial. Quand O'Reilly définit « open source » comme modèle de développement supérieur plutôt que comme position éthique, il ne décrit pas le monde — il le reconfigure. Les entreprises qui adoptent ensuite le terme héritent d'un cadre conceptuel où la liberté de l'utilisateur n'est simplement plus un sujet. C'est le mécanisme par lequel l'extraction devient invisible : non pas en la cachant, mais en changeant le vocabulaire au point que la question ne se pose plus.

2.4 L'extraction corporative précoce

IBM et le pari du milliard

[FAIT] En 2001, IBM a annoncé un investissement d'un milliard de dollars dans Linux, couvrant le développement, le marketing, les ventes, le support ISV et la publicité mondiale. L'IBM Linux Technology Center, créé en 1999 avec 185 employés, comptait environ 600 personnes en 2006, dont 300 travaillant à temps plein sur Linux. Le vice-président Bill Zeitler a déclaré avoir récupéré la majeure partie de l'investissement dès la première année en ventes de logiciels et de systèmes. 🟢 90-100%

[INTERPRÉTATION] L'analyste Dan Kusnetzky d'IDC a souligné qu'IBM avait probablement inclus dans ce chiffre des investissements destinés à d'autres systèmes d'exploitation (AIX, UnixWare), créant délibérément les conditions pour revendiquer un montant massif. Kusnetzky reconnaît cependant que la stratégie était brillante : IBM tirait davantage parti de Linux et du mouvement open source que n'importe lequel de ses concurrents. 🟡 70-89%

[INTERPRÉTATION] Le modèle IBM était le suivant : investir dans une infrastructure commune (Linux) pour éliminer les coûts de licence payés à des concurrents (Sun, Microsoft), tout en vendant du matériel, du middleware et des services propriétaires au-dessus. C'était symbiotique dans la mesure où IBM contribuait réellement au noyau Linux. C'était extractif dans la mesure où la valeur créée par des milliers de contributeurs bénévoles était capturée dans une offre commerciale dont le bénéfice revenait aux actionnaires d'IBM. Les deux étaient vrais simultanément.

Le regard des mainteneurs

[FAIT] La vulnérabilité Heartbleed, divulguée en 2014, a révélé que la bibliothèque OpenSSL — utilisée par la majorité des

serveurs web pour sécuriser le trafic Internet — était maintenue par une équipe minuscule, avec un seul développeur à temps plein et un salaire bien inférieur aux normes de l'industrie. Avant Heartbleed, le projet ne recevait pas plus de 2 000 dollars de dons par an. 🟢 90-100%

[FAIT] Nadia Eghbal, dans son rapport de 2016 pour la Ford Foundation intitulé « Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure », a documenté que l'écrasante majorité des projets open source sont maintenus par une ou deux personnes. Selon une enquête Tidelift de 2020, 46% des mainteneurs professionnels ont connu le burnout, et ce chiffre monte à 58% pour les mainteneurs de projets très utilisés. 🟡 70-89%

[INTERPRÉTATION] L'expression de Kyle Mitchell, fondateur de License Zero, résume la situation : il existe une hypothèse selon laquelle l'open source continuera à tomber du ciel comme la manne et que les personnes derrière les projets peuvent être abstraites. Cette abstraction est structurelle : le recadrage « open source », en évacuant le langage des droits et de la liberté, a aussi évacué la question de ce qui est dû aux personnes qui maintiennent l'infrastructure.

2.5 Les enjeux humains

Identité et artisanat

[INTERPRÉTATION] L'ère artisanale a produit un type de développeur dont l'identité professionnelle reposait sur la maîtrise totale de ses outils. Comprendre son logiciel, c'était le posséder intellectuellement — même si on ne le possédait pas juridiquement. Le mouvement du logiciel libre a donné à cette maîtrise une expression juridique (la licence) et

communautaire (le partage). Le recadrage open source a conservé la communauté mais a détaché la maîtrise de la liberté : on pouvait désormais utiliser du code ouvert sans s'engager dans aucun projet d'émancipation.

Communauté et appartenance

[FAIT] Les communautés du logiciel libre des années 1990-2000 étaient des communautés de pratique authentiques : listes de diffusion, IRC, conférences, hackerspaces. Elles étaient aussi profondément exclusives sur des axes de genre, de langue et de géographie. La culture dominante était anglophone, masculine, et ancrée dans les universités américaines et les entreprises de la côte Ouest. Les contributeurs d'Amérique latine, d'Afrique ou d'Europe non anglophone accédaient à ces espaces, mais rarement aux positions d'influence. 🟡 70-89%

Signaux politiques précoces

[FAIT] Plusieurs gouvernements ont reconnu la dimension politique des chaînes d'approvisionnement logicielles dès cette période. La France a tenté dès 1999 une proposition de loi pour généraliser le libre dans l'administration (restée sans suite). Le Brésil a commencé à adopter des lois locales dès 1999-2001 (Amparo, Recife). L'Italie a modifié son code des marchés publics en 2012 pour imposer une analyse comparative avant tout achat propriétaire. Le Pérou a adopté la loi 28612 après la célèbre réponse du député Edgar Villanueva à Microsoft en 2002. 🟡 70-89%

---

3. Contradictions et signaux faibles

Le recadrage : trahison ou compromis nécessaire ?

La thèse centrale de cette enquête — que le recadrage « open source » a rendu l'extraction culturellement acceptable — mérite d'être testée honnêtement.

Argument pour la trahison : Stallman avait raison sur le diagnostic. Le vocabulaire de l'efficacité commerciale a effectivement détourné l'attention des questions de liberté. Vingt-cinq ans plus tard, les géants de la technologie utilisent massivement du code libre sans que la question de la liberté des utilisateurs ne soit jamais posée. Le logiciel est « open » mais l'utilisateur est enfermé dans des plateformes.

Argument pour le compromis nécessaire : Le recadrage a massivement élargi la participation. Sans l'investissement d'IBM, de Sun, de Google, Linux ne serait probablement pas devenu l'infrastructure de l'Internet. Les outils produits ont été réellement utiles à des millions de développeurs individuels. Le marché de l'open source en France pèse 5 milliards d'euros et emploie des milliers de personnes.

Signal faible : Le fait que Perens, cofondateur de l'OSI, ait lui-même critiqué la cooptation du mouvement par des intérêts commerciaux suggère que les deux lectures ne sont pas mutuellement exclusives. Le recadrage était un compromis pragmatique qui a aussi, et simultanément, ouvert la porte à une extraction systématique. Les deux choses sont vraies. La tension n'est pas résoluble — elle est constitutive.

Le silence sur le travail de maintenance

La contradiction la plus grave révélée par cette recherche est le décalage entre la rhétorique de communauté et la réalité du travail. L'open source est célébré comme modèle de collaboration ; en pratique, il repose sur un petit nombre de personnes souvent non payées, surmenées, et dont le travail est

structurellement invisible. Le cas Heartbleed n'est pas une anomalie — c'est le fonctionnement normal du système.

---

4. Angles morts

Genre et diversité. Cette recherche n'a pas suffisamment exploré les dynamiques d'exclusion au sein des communautés du logiciel libre. L'histoire canonique est racontée presque exclusivement par des hommes blancs américains. Les contributions des femmes, des personnes non anglophones et des développeurs du Sud global sont structurellement sous-documentées.

Asie et monde sinophone. Le rôle de la Chine, du Japon et de la Corée dans l'adoption et la transformation du modèle open source n'a pas été couvert. La dimension géopolitique de la chaîne d'approvisionnement logicielle dans le contexte sino-américain mériterait une session dédiée.

Monde arabe et MENA. Aucune source n'a été trouvée sur les communautés du logiciel libre dans les pays arabes durant cette période. C'est probablement un angle mort de la recherche plutôt qu'une absence réelle.

Les licences comme terrain de lutte. La bataille GPL vs. licences permissives (BSD, MIT, Apache) est un terrain riche qui n'a été qu'effleuré ici. Le choix de licence est un acte politique qui détermine précisément les conditions d'extraction.

---

5. Synthèse actionnable

L'ère 1 de la chaîne d'approvisionnement logicielle établit cinq mécanismes qui rendront l'extraction de l'ère suivante

possible :

Premier mécanisme : le recadrage linguistique. En remplaçant « liberté » par « ouverture », le mouvement a perdu la capacité de nommer ce qui était pris. L'extraction sans nom est l'extraction la plus efficace.

Deuxième mécanisme : la séparation entre production et maintenance. L'open source a établi un modèle où la création de code est valorisée (contributions, réputation, embauche) tandis que la maintenance est invisible. Ce déséquilibre est la condition structurelle du burnout et de la vulnérabilité de l'infrastructure.

Troisième mécanisme : l'investissement comme légitimation. Quand IBM investit un milliard de dollars dans Linux, le message est : l'open source est sérieux, légitime, digne de l'entreprise. Ce message est vrai. Il normalise aussi l'idée que le commun est un input pour la création de valeur privée — pas une fin en soi.

Quatrième mécanisme : la communauté comme ressource. Les communautés du logiciel libre ont été progressivement reconfigurées de communautés de pratique en viviers de main-d'œuvre. Le développeur qui contribue à un projet open source construit son CV autant qu'il construit un commun. Quand la motivation principale devient le signal de compétence pour l'embauche, la communauté devient un marché du travail déguisé.

Cinquième mécanisme : l'invisibilisation des alternatives. Le succès du terme « open source » a marginalisé le vocabulaire du logiciel libre. La question de la liberté de l'utilisateur est devenue une position minoritaire, presque excentrique, alors qu'elle était le point de départ du mouvement. Les alternatives

politiques au modèle d'extraction existent — elles sont simplement devenues difficiles à nommer.

---

6. Pistes d'approfondissement

Session 2 (prévue) : L'ère des plateformes. Comment GitHub, npm, Docker et l'écosystème des gestionnaires de paquets ont transformé le commun logiciel en pipeline extractif. Le rôle de la dépendance transitive, de l'effet de réseau, et de la centralisation sur des plateformes privées.

Session 3 (prévue) : L'ère générative. Comment l'entraînement de modèles de langage sur des corpus de code open source constitue la forme la plus radicale d'extraction à ce jour : le code est transformé en poids statistiques qui remplacent le développeur lui-même. La commodification de l'écriture de code.

Approfondissement nécessaire : le rôle des licences comme terrain de lutte (GPL vs. permissives), la dimension genre dans l'histoire du libre, le rôle de la Chine dans la transformation du modèle open source, et les expériences européennes de logiciel public (ADULLACT en France, publiccode.eu).

---

7. Pont vers l'ère 2

L'ère 1 a ouvert cinq portes que l'ère 2 a traversées. Le recadrage linguistique a rendu l'extraction nommable comme « innovation ». La séparation production/maintenance a créé une infrastructure fragile et gratuite que les plateformes ont pu encapsuler. L'investissement corporatif a légitimé l'idée que le commun est un input, pas un bien en soi. La transformation des communautés en viviers de main-d'œuvre a préparé le

terrain pour les marchés du travail algorithmiques. Et l'invisibilisation du vocabulaire de la liberté a rendu presque impossible de nommer ce qui était perdu.

Quand GitHub rachète la distribution du code, quand npm centralise les dépendances, quand Docker encapsule l'environnement d'exécution — ces mouvements ne sont possibles que parce que l'ère 1 a établi qu'il était normal, acceptable, et même souhaitable de construire une entreprise au-dessus d'un commun sans poser la question de ce qui revient au commun.

> *La route est longue, mais la voie est libre.* — Proverbe du mouvement du logiciel libre

---

8. Table des sources

#	Source	Type	Date	Langue
1	Christine Peterson, « How I Coined the Term Open Source Software », Opensource.com	📝 Blog	02/2018	EN
2	R. Stallman, « The GNU Operating System and the Free Software Movement », O'Reilly Open Sources	🏛️ Primaire	1999	EN
3	S. Broca, « Retour sur les enjeux politiques du mouvement du logiciel libre », OpenEdition Books	🎓 Académique	2020	FR
4	A. Shaw, « Insurgent Expertise: The Politics of FLOSS in Brazil », JITP	🎓 Académique	2011	EN

#	Source	Type	Date	Langue
5	B. Birkinbine, « Free Software as Public Service in Brazil », IJOC	🎓 Académique	2016	EN
6	Linux.com, « Brazilian President Lula brings attention to Free Software »	📰 Presse spéc.	07/2009	EN
7	ADULLACT, Tribune « Les logiciels libres ont besoin d'une stratégie publique »	🏛️ Primaire	2021	FR
8	Labo Société Numérique, « Les logiciels libres et open source en France »	🏛️ Primaire	s.d.	FR
9	Acteurs Publics, « Logiciels libres dans l'Administration »	📰 Presse spéc.	11/2021	FR
10	STRATPOL, « La bataille pour le logiciel libre en Afrique »	📰 Presse spéc.	11/2021	FR
11	Financial Afrik, « Pourquoi l'Afrique doit privilégier les logiciels libres »	📰 Presse spéc.	07/2025	FR
12	RAFLL, Site officiel des Rencontres Amicales Francophones du Logiciel Libre	🏛️ Primaire	2026	FR
13	E.S. Raymond, « The Open-Source Movement: 1998 and Onward », catb.org	🏛️ Primaire	2003	EN

#	Source	Type	Date	Langue
14	R. Poynder, « Open and Shut?: Interview with Eric Raymond »	📝 Blog	03/2006	EN
15	Bruce Perens, Archania wiki, histoire complète	📝 Blog	s.d.	EN
16	Wikipedia (EN), « Linux Technology Center »	📝 Encyclopédie	01/2025	EN
17	HPC Wire, « IBM: Linux Investment Nearly Recouped »	📰 Presse spéc.	02/2002	EN
18	practical-tech.com, « IBM and Linux: The Early Years »	📰 Presse spéc.	05/2002	EN
19	N. Eghbal, « Roads and Bridges », Ford Foundation	🏛️ Primaire	2016	EN
20	N. Eghbal, « Working in Public », Stripe Press	🏛️ Primaire	2020	EN
21	TechCrunch, « Open source sustainability »	📰 Presse spéc.	06/2018	EN
22	The Register, « When software depends on a project maintained by a random guy in Nebraska »	📰 Presse spéc.	05/2021	EN
23	Libre à Lire, « Le logiciel libre, une lutte politique — PSES 2024 »	🏛️ Primaire	08/2024	FR
24	OSI, « President Lula receives ITU Award, Open Source cited »	🏛️ Primaire	06/2009	EN

#	Source	Type	Date	Langue
25	Wikipedia (FR), « Mouvement du logiciel libre »	📝 Encyclopédie	07/2025	FR

Recherche effectuée en mars 2026. Langues : français, anglais, portugais (Brésil). Toutes les URL visitées pendant la recherche ; certaines omises de la table pour brièveté mais disponibles sur demande.

← L'enclosure du code — Série complète

floss - 2 - L'enclosure du code: l'ère de l'assemblage

2026 / 04 / 19 · https://hyper.dev/2026/enclosure-du-code/2-ere-de-assemblage/

Comment l'extraction de valeur à partir des communs logiciels est devenue invisible — et ce que cela a fait aux personnes qui écrivent du code

Deuxième session — La chaîne d'approvisionnement logicielle, des années 2010 à l'ère pré-générative

← L'enclosure du code — Série complète

---

1. Synthèse

[INTERPRÉTATION] Entre 2010 et 2022, le logiciel a cessé d'être quelque chose qu'on construit pour devenir quelque chose qu'on assemble. Les gestionnaires de paquets (npm, pip, Maven) ont transformé la production logicielle en un réseau de dépendances transitives si vaste que personne ne pouvait le gouverner. 🟢 90-100% Les plateformes cloud (AWS, GCP, Azure) sont devenues les nouveaux gardiens : elles n'hébergeaient pas simplement le logiciel — elles *étaient* la chaîne d'approvisionnement. 🟢 90-100% Deux incidents — left-pad (2016) et Log4Shell (2021) — ont rendu brièvement visible le travail invisible des mainteneurs non payés qui soutenaient cette infrastructure. 🟢 90-100%

Cette ère a normalisé l'extraction au point de l'invisibilité. Le modèle *open core* financé par le capital-risque a fait de la capture de valeur à partir des communs une stratégie d'affaires explicite et célébrée. 🟡 70-89% Simultanément — et c'est la

tension qu'il faut tenir — cette même ère a représenté la plus grande démocratisation de la capacité à construire du logiciel de l'histoire humaine. Un développeur seul à Lagos, Dakar ou Tizi Ouzou pouvait expédier un produit qui aurait requis une équipe de dix personnes en 2005. Les deux choses sont vraies. La question est de savoir laquelle a laissé le plus de traces.

---

2. Résultats détaillés

2.1 L'explosion des dépendances : quand le logiciel est devenu un graphe

Le passage de « écrire » à « trouver »

[FAIT] npm, le gestionnaire de paquets pour JavaScript, a été lancé en 2010. En 2016, il hébergeait plus de 250 000 paquets. En 2025, le registre npm dépasse les 2 millions de paquets. PyPI (Python), Maven Central (Java), RubyGems et NuGet ont suivi des trajectoires similaires. 🟢 90-100%

[INTERPRÉTATION] Ce que ces chiffres décrivent n'est pas simplement une expansion. C'est un changement de nature dans ce que signifie « développer du logiciel ». Quand un développeur installe un framework front-end comme React ou Angular, il ne charge pas un outil — il charge un arbre de dépendances qui peut inclure des centaines, voire des milliers de paquets écrits par des personnes qu'il ne connaît pas, qu'il ne lira jamais, et dont il dépend entièrement. Le blogueur David Haney a découvert après l'incident left-pad qu'une installation fraîche de Babel incluait 41 000 fichiers. Un paquet appelé `isArray` — une seule ligne de code — était téléchargé 880 000 fois par jour.

L'analogie est celle de la grande distribution : comme un hypermarché qui assemble des produits de milliers de

fournisseurs sur un linéaire que le consommateur parcourt sans voir la chaîne d'approvisionnement, le développeur moderne assemble du code qu'il n'a pas inspecté, dans un graphe de dépendances qu'il ne peut pas gouverner.

left-pad : onze lignes qui ont cassé Internet (2016)

[FAIT] Le 22 mars 2016, Azer Koçulu, un développeur basé à Oakland, a retiré de npm les 273 paquets qu'il avait publiés, dont `left-pad` — une fonction de 11 lignes qui ajoute des caractères au début d'une chaîne de texte. Des milliers de projets, dont Babel, Webpack et React, dépendaient de left-pad directement ou transitivement. Facebook, PayPal, Netflix, Spotify ont été affectés. Le paquet avait été téléchargé 2,5 millions de fois dans le mois précédant sa suppression. npm a restauré le paquet deux heures plus tard — un acte sans précédent. 🟢 90-100%

[FAIT] Koçulu avait agi en protestation contre npm Inc., qui avait transféré le nom de paquet « kik » à l'entreprise Kik Interactive après que celle-ci eut invoqué sa marque commerciale. Koçulu a écrit : « Cette situation m'a fait réaliser que npm est le terrain privé de quelqu'un, où les entreprises sont plus puissantes que les personnes. » 🟢 90-100%

[INTERPRÉTATION] L'incident est révélateur à trois niveaux. D'abord, il a montré que des entreprises valant des milliards de dollars dépendaient de onze lignes de code écrites par un développeur individuel, sans contrat, sans rémunération, sans même en être conscientes. Ensuite, la réponse de npm — restaurer un paquet contre la volonté de son auteur — a démontré que dans un système de dépendances massives, la propriété individuelle du développeur cède devant les « besoins du plus grand nombre ». Enfin, comme l'a observé

Armin Ronacher, l'approche « micro-dépendance » de npm crée une surface d'attaque qu'aucune personne raisonnable ne peut inspecter. Un développeur individuel maintenait 827 modules séparés sur npm. 🟡 70-89%

Log4Shell : quand le commun s'effondre à l'échelle planétaire (2021)

[FAIT] En décembre 2021, la vulnérabilité Log4Shell (CVE-2021-44228) a été divulguée dans Apache Log4j, une bibliothèque Java de journalisation utilisée par la majorité des serveurs d'entreprise dans le monde. La vulnérabilité existait inaperçue depuis 2013. Apache lui a attribué un score CVSS de 10 — le maximum. La CISA a décrit Log4Shell comme potentiellement la vulnérabilité la plus grave jamais vue. Des centaines de millions d'appareils étaient affectés, y compris des systèmes d'Apple, Amazon, Cloudflare, Tesla, Twitter et Baidu. 🟢 90-100%

[FAIT] Log4j était maintenu par une poignée de bénévoles non payés. La FTC américaine a reconnu que ces projets sont « souvent créés et maintenus par des volontaires qui n'ont pas toujours les ressources adéquates pour la réponse aux incidents et la maintenance proactive, même quand leurs projets sont critiques ». L'Agence de cybersécurité de Singapour a comparé Log4j à « un type de vis largement utilisé dans les produits semi-finis et finis — moteurs, chaises, tables, automobiles — toute vulnérabilité dans un tel composant fondamental de la chaîne d'approvisionnement logicielle peut avoir des conséquences profondes ». 🟢 90-100%

[INTERPRÉTATION] Log4Shell n'est pas un incident isolé. C'est le fonctionnement normal d'un système où l'infrastructure critique est maintenue par des bénévoles. Le schéma est identique à Heartbleed (2014, OpenSSL) et à

l'incident colors.js/faker.js (2022, où le mainteneur Marak Squires a délibérément saboté ses propres paquets en protestation contre le travail gratuit). Ces incidents ne sont pas des bugs — ce sont des révélations : des moments où la réalité du travail invisible devient brièvement visible avant que le système ne la recouvre. 🟡 70-89%

2.2 Les plateformes cloud comme nouveaux gardiens

« Vous le construisez, nous vous le relouons »

[FAIT] Amazon Web Services a lancé son service Amazon Elasticsearch en 2015, utilisant le projet open source Elasticsearch sous licence Apache 2.0. Elastic, l'entreprise qui avait créé et maintenait le projet, a protesté que ce service avait été lancé « sans collaborer avec nous » et qu'il créait de la confusion sur la responsabilité des bugs. En 2021, Elastic a changé sa licence vers la SSPL (Server Side Public License), conçue pour empêcher les fournisseurs cloud d'offrir le logiciel comme service sans contribuer en retour. AWS a répondu en forkant le projet sous le nom OpenSearch, maintenant la licence Apache 2.0. 🟢 90-100%

[FAIT] Le même schéma s'est répété : MongoDB a adopté la SSPL en 2018 pour se protéger d'AWS. Redis a changé de licence BSD vers SSPL/RSALv2 en mars 2024 ; la Linux Foundation a immédiatement annoncé Valkey, un fork soutenu par AWS, Google et d'autres. CockroachDB et Confluent (Apache Kafka) ont effectué des changements de licence similaires entre 2019 et 2020. 🟢 90-100%

[INTERPRÉTATION] Ces conflits ne sont pas des disputes commerciales ordinaires. Ce sont des moments où la communauté a débattu de la question : *qui possède ce que les*

communs produisent ? D'un côté, Elastic et MongoDB argumentaient que les fournisseurs cloud capturaient la valeur créée par leurs développeurs sans rien retourner. De l'autre, AWS argumentait qu'elle défendait l'open source en maintenant des forks sous licence permissive. Les deux positions sont sincères. Mais la dynamique structurelle est claire : quand le fournisseur cloud peut forker n'importe quel projet open source et l'offrir comme service géré, le pouvoir ne réside plus dans le code — il réside dans l'infrastructure. C'est la même dynamique qui permet à un hypermarché d'imposer ses conditions à ses fournisseurs : celui qui contrôle le linéaire dicte les termes. 🟡 70-89%

2.3 L'extraction comme modèle d'affaires célébré

[INTERPRÉTATION] Pendant cette ère, le capital-risque a explicitement financé la stratégie de construction sur des fondations open source tout en capturant la valeur en privé. Le modèle « open core » — un noyau open source gratuit surmonté de fonctionnalités propriétaires payantes — est devenu le playbook standard. MongoDB l'a décrit dans son introduction en bourse (S-1, 2017) : la version gratuite existe « pour encourager l'usage par les développeurs, la familiarité et l'adoption de notre plateforme ». C'est une stratégie de dépendance délibérée : l'open source crée l'addiction, la version payante capture la valeur. 🟡 70-89%

[FAIT] Les projets ayant changé de licence — MongoDB, Elastic, Redis, CockroachDB, Confluent — sont tous contrôlés par des entreprises uniques, généralement adossées à du capital-risque. Le capital-risque cherche des options de sortie (introduction en bourse, acquisition). Cette tension entre la logique communautaire de l'open source et la logique de sortie du capital-risque est structurelle, pas accidentelle. 🟡 70-89%

[INTERPRÉTATION] La reconnaissance de ce schéma par la communauté des développeurs a été progressive plutôt que soudaine. Des voix critiques existaient tôt — le mainteneur de Typesense a noté sur Hacker News que l'open source est « formidable pour attirer des contributeurs » mais devient « un albatros autour du cou » quand il s'agit de générer des revenus. Mais la prise de conscience collective s'est cristallisée autour des incidents concrets : les changements de licence, les forks par les géants du cloud, les burnouts publics de mainteneurs.

2.4 La « classe de configuration » : quand développer devient assembler

La fatigue des frameworks

[FAIT] L'enquête Stack Overflow 2024 (65 437 répondants de 185+ pays) révèle que le burnout n'est plus subtil — il est bruyant. Pour la première fois, les développeurs seniors rapportent une satisfaction professionnelle plus basse que les juniors. Une enquête Stack Overflow de 2022 a trouvé que 2 développeurs sur 5 présentent un risque élevé de burnout. 🟡 70-89%

[INTERPRÉTATION] Le phénomène de « JavaScript fatigue » ou « framework fatigue » a un nom depuis au moins 2016. Le cycle d'adoption, d'apprentissage et de migration entre frameworks crée ce qu'un analyste décrit comme « un burnout silencieux même parmi les équipes les plus dévouées ». Le problème n'est pas simplement qu'il y a trop d'outils — c'est que chaque changement de pile technologique impose un coût cognitif et temporel : configuration, débogage, nouvelles abstractions à comprendre. Le développeur moderne ne code pas tant qu'il ne configure : il assemble des services, relie des API, et gère des fichiers de configuration pour des outils qu'il n'a pas choisis et ne comprend pas pleinement.

David Haney posait la question dès 2016 : « Si vous ne pouvez pas écrire une fonction left-pad en 5 minutes, vous ne savez pas coder. Enfiler des appels d'API les uns aux autres et appeler ça de la programmation n'en fait pas de la programmation. C'est une forme délirante de piratage de dépendances impliquant le cloud, la sur-ingénierie et une complexité bien au-delà de ce qui est nécessaire. »

L'érosion de l'identité artisanale

[INTERPRÉTATION] Quand le travail passe de « construire » à « assembler », quelque chose se perd dans l'identité professionnelle. Le syndrome de l'imposteur dans les communautés de développeurs est omniprésent : le sentiment de ne pas « vraiment » comprendre la pile technologique qu'on utilise, l'anxiété de dépendre de code qu'on n'a jamais lu. L'ère artisanale avait produit un développeur dont la dignité venait de la maîtrise. L'ère de l'assemblage produit un développeur dont la productivité vient de la vitesse de configuration — et dont la dignité est menacée par la prochaine mise à jour de framework qui rend ses compétences obsolètes.

Le contraste avec l'ère 1 est frappant : le développeur des années 1990 savait ce que faisait chaque composant de sa pile. Le développeur de l'ère de l'assemblage gère un graphe de dépendances qu'aucune personne ne peut inspecter. Le premier avait une relation artisanale avec ses outils. Le second a une relation de dépendance avec ses fournisseurs — invisibles, non payés, et potentiellement en burnout.

2.5 Les enjeux humains

Le réveil politique : quand les gouvernements découvrent la chaîne d'approvisionnement

[FAIT] L'Executive Order 14028 du président Biden (mai 2021) a rendu les Software Bills of Materials (SBOM) obligatoires pour les fournisseurs du gouvernement fédéral américain. Le EU Cyber Resilience Act (CRA), proposé en 2022 et adopté en 2024 (Règlement 2024/2847), impose des exigences de cybersécurité obligatoires pour tous les fabricants vendant des produits numériques sur le marché européen, avec entrée en vigueur complète en décembre 2027. Le CRA exige explicitement un SBOM dans la documentation technique et la capacité de rappeler des produits défectueux — allant plus loin que l'approche américaine. 🟢 90-100%

[INTERPRÉTATION] Ces réponses politiques sont significatives mais tardives. Elles arrivent après deux décennies pendant lesquelles la chaîne d'approvisionnement logicielle s'est construite sans supervision. L'approche CRA — qui impose non seulement la transparence mais la capacité de rappel — est structurellement plus ambitieuse que l'approche américaine. Mais un acteur est absent de la conversation : les mainteneurs eux-mêmes. Comme le note Sonatype, le CRA pourrait être « bon pour la sécurité de la chaîne d'approvisionnement, mauvais pour l'open source » — en imposant des obligations de conformité à des bénévoles qui n'ont déjà pas les ressources pour maintenir leurs projets. 🟡 70-89%

L'asymétrie globale

[INTERPRÉTATION] L'ère de l'assemblage a démocratisé l'accès aux outils de développement. Un développeur à Abidjan installe les mêmes paquets npm qu'un développeur à

San Francisco. Mais l'asymétrie persiste : moins de bande passante pour télécharger les dépendances, moins d'infrastructure pour les tests, et aucun siège à la table de gouvernance. Les communautés francophones du logiciel libre (les RAFLL, Ovillage en Côte d'Ivoire, les communautés OpenStreetMap en Afrique de l'Ouest) continuent de construire des alternatives, mais elles opèrent à l'intérieur d'un écosystème de dépendances qu'elles ne contrôlent pas.

Le paradoxe est complet : npm, pip et Maven ont donné au monde entier les mêmes briques de base. Mais la gouvernance de ces briques — les décisions sur les licences, les politiques de suppression, les priorités de maintenance — reste concentrée dans une poignée d'entreprises américaines. La démocratisation de l'usage n'est pas la démocratisation du pouvoir.

---

3. Contradictions et signaux faibles

L'assemblage : asservissement ou émancipation ?

Argument pour l'asservissement : L'ère de l'assemblage a créé une dépendance structurelle envers des communs non financés, des plateformes cloud oligopolistiques, et un écosystème de frameworks en rotation permanente. Le développeur individuel a perdu la maîtrise de ses outils au profit de la vitesse de livraison. Le burnout est devenu endémique. L'extraction est devenue invisible.

Argument pour l'émancipation : Cette même ère a permis à plus de personnes de construire du logiciel que jamais dans l'histoire. Un développeur autodidacte avec un ordinateur et une connexion Internet peut expédier un produit fonctionnel en quelques semaines. L'enquête Stack Overflow 2024 montre

que les développeurs n'attendent plus la permission d'une institution pour apprendre — « ils trouvent la documentation, vont sur YouTube, et apprennent en public ». Les barrières d'infrastructure ont été écrasées par le cloud. Les barrières de connaissance ont été écrasées par les gestionnaires de paquets.

Le signal faible : Les deux sont vrais, mais ils ne sont pas vrais pour les mêmes personnes. L'émancipation profite au développeur qui assemble un produit pour expédier. L'asservissement frappe le mainteneur qui maintient les briques que tout le monde utilise. Le premier est célébré. Le second est invisible. C'est la même structure que dans la grande distribution : le consommateur bénéficie de prix bas ; le producteur est écrasé par la pression sur les marges. La question n'est pas « est-ce que c'est bon ou mauvais ? » mais « pour qui ? ».

Le silence autour du modèle d'affaires du cloud

Le débat Elastic/AWS a été traité par la presse spécialisée comme un conflit commercial entre entreprises. Il est rarement formulé dans les termes qui lui conviendraient : un débat sur l'enclosure des communs numériques. Quand un fournisseur cloud peut forker n'importe quel projet open source et l'offrir comme service géré, les incitations à contribuer au commun s'effondrent pour tous ceux qui ne sont pas le fournisseur cloud. C'est le mécanisme classique de la tragédie des communs — mais avec cette différence que la tragédie est organisée, financée et célébrée comme innovation.

---

4. Angles morts

L'économie politique du capital-risque dans l'open source. Les dynamiques de pouvoir entre investisseurs, fondateurs et

communautés dans les entreprises open core méritent une analyse dédiée. La pression vers la sortie (IPO, acquisition) déforme systématiquement les incitations communautaires.

Les développeurs du Sud global. Cette recherche manque de données primaires sur l'expérience vécue par les développeurs en Afrique, en Asie du Sud et en Amérique latine face à la dépendance aux plateformes. Les RAFLL et les communautés similaires sont documentées, mais l'expérience quotidienne du développeur qui assemble des paquets npm depuis Dakar ou Dhaka reste sous-étudiée.

Les alternatives concrètes. Deno, Bun, et d'autres projets tentent de réduire l'explosion des dépendances. Les « vendored dependencies » (copie des dépendances dans l'arbre source) sont une pratique ancienne qui connaît un regain d'intérêt. Ces alternatives n'ont pas été explorées ici.

Le genre et la diversité dans la maintenance. Qui sont les mainteneurs ? Les données d'Eghbal et de Tidelift montrent que ce sont souvent des individus isolés, mais l'analyse genrée et géographique de cette population est insuffisante.

---

5. Synthèse actionnable

L'ère de l'assemblage a établi quatre conditions qui rendent l'ère générative (ère 3) non seulement possible mais prévisible :

Première condition : un corpus massif de code sur GitHub. L'explosion des dépendances et la centralisation du code sur GitHub (racheté par Microsoft en 2018) ont créé le plus grand corpus de code jamais assemblé. Ce corpus est la matière

première sur laquelle les modèles de langage ont été entraînés. Sans l'ère de l'assemblage, il n'y aurait pas de Copilot.

Deuxième condition : une main-d'œuvre habituée à assembler plutôt qu'à construire. La « classe de configuration » — les développeurs dont le travail consiste à câbler des services plutôt qu'à écrire des algorithmes — est la population la plus immédiatement menacée par les outils de génération de code. Si votre travail est déjà d'assembler des appels d'API, un modèle de langage peut le faire aussi.

Troisième condition : une attente normalisée que les communs existent pour être moissonnés. Le recadrage « open source » de l'ère 1, renforcé par le modèle open core de l'ère 2, a normalisé l'idée que le code des communs est un input pour la création de valeur privée. L'entraînement de modèles de langage sur des corpus de code open source est la conclusion logique de cette normalisation.

Quatrième condition : un paysage politique qui commence à peine à comprendre les enjeux. L'Executive Order de 2021 et le CRA européen sont des réponses à des incidents passés (SolarWinds, Log4Shell). Ils ne sont pas conçus pour les enjeux de l'ère générative — l'extraction de savoir-faire à partir du corpus GitHub, la commodification de l'écriture de code, la question de ce qui revient aux auteurs du code sur lequel les modèles sont entraînés.

---

6. Pistes d'approfondissement

Session 3 (prévue) : L'ère générative. Comment l'entraînement de modèles de langage sur des corpus de code open source constitue la forme la plus radicale d'extraction à ce jour. Le code est transformé en poids statistiques qui remplacent le

développeur lui-même. La question des droits d'auteur, la commodification de l'écriture de code, et l'impact sur l'identité professionnelle des développeurs.

Approfondissement nécessaire : l'économie politique du capital-risque dans l'open source ; l'expérience vécue des développeurs du Sud global face à la dépendance aux plateformes ; les alternatives concrètes au modèle de dépendances massives (vendored dependencies, Deno, Guix, Nix) ; la gouvernance des fondations open source (Apache, Linux Foundation, CNCF) et leur relation aux entreprises qui les financent.

---

7. Pont vers l'ère 3

L'ère 2 a laissé sur la table exactement ce que l'ère 3 avait besoin de trouver.

Un corpus. GitHub héberge plus de 200 millions de dépôts. La quasi-totalité du code public produit pendant l'ère de l'assemblage est indexé, accessible, et stocké dans un format que les modèles de langage peuvent ingérer. Ce n'est pas un accident — c'est la conséquence directe du choix de centraliser le développement logiciel mondial sur une plateforme unique, rachetée par Microsoft pour 7,5 milliards de dollars en 2018.

Une main-d'œuvre. Des millions de développeurs dont le travail quotidien consiste déjà à assembler plutôt qu'à construire — et qui sont donc les premiers candidats à la commodification par des outils qui assemblent plus vite qu'eux.

Une norme. L'idée que le code produit dans les communs est un input librement disponible pour la création de valeur

privée. Si IBM pouvait construire un milliard de dollars de ventes sur Linux, si AWS pouvait forker Elasticsearch et l'offrir comme service, alors pourquoi OpenAI ne pourrait-il pas entraîner un modèle sur l'intégralité de GitHub ?

Un vide politique. Les régulateurs commencent à peine à comprendre les SBOM et les chaînes de dépendances. La question de savoir ce qui revient aux auteurs du code sur lequel les modèles sont entraînés n'a pas encore de cadre juridique. Quand l'ère générative arrive, elle trouve un terrain sans clôtures.

> *Le logiciel est « open » mais l'utilisateur est enfermé dans des plateformes. La route est libre — mais le péage est à la sortie.*

---

8. Table des sources

#	Source	Type	Date	Langue
1	Wikipedia, « npm left-pad incident »	📝 Encyclopédie	2026	EN
2	The Register, « How one developer just broke Node, Babel and thousands of projects »	📰 Presse spéc.	03/2016	EN
3	Quartz, « How one programmer broke the internet by deleting a tiny piece of code »	📰 Presse gén.	03/2016	EN
4	D. Haney, « NPM & left-pad: Have We Forgotten How To Program? »	📝 Blog	03/2016	EN

#	Source	Type	Date	Langue
5	npm Blog, « kik, left-pad, and npm » (post-mortem officiel)	🏛️ Primaire	03/2016	EN
6	LWN.net, « A single Node of failure »	📰 Presse spéc.	03/2016	EN
7	Wikipedia, « Log4Shell »	📝 Encyclopédie	12/2025	EN
8	CISA, « Apache Log4j Vulnerability Guidance »	🏛️ Primaire	12/2021	EN
9	CISA/FBI/NSA, « Mitigating Log4Shell and Other Log4j-Related Vulnerabilities »	🏛️ Primaire	12/2021	EN
10	PMC/NIH, « Log Jam: Lessons Learned from the Log4Shell Vulnerability »	🎓 Académique	2023	EN
11	DataCenter Knowledge, « Log4Shell Vulnerability Highlights Software Supply Chain Issues »	📰 Presse spéc.	01/2022	EN
12	CSA Singapore, « The Log4Shell Vulnerability »	🏛️ Primaire	2022	EN
13	Wikipedia, « Server Side Public License »	📝 Encyclopédie	01/2026	EN
14	TechTarget, « Elastic vs. AWS highlights open source monetization dilemma »	📰 Presse spéc.	2021	EN
15	The Register, « OpenSearch, the AWS-	📰 Presse spéc.	07/2021	EN

#	Source	Type	Date	Langue
	sponsored Elasticsearch fork, reaches 1.0 »			
16	Computing.co.uk, « Elastic has stretched the patience of many in open source »	📰 Presse spéc.	01/2021	EN
17	Architecture Weekly, « Why Open Source Isn't Always Fair »	📝 Blog	08/2025	EN
18	Stack Overflow, « 2024 Developer Survey »	🏛️ Primaire	05/2024	EN
19	ShiftMag, « Stack Overflow Survey: 80% of developers are unhappy »	📰 Presse spéc.	01/2025	EN
20	Medium/Devglyph, « How Chasing Every New JavaScript Framework Burned Me Out »	📝 Blog	10/2025	EN
21	N. Eghbal, « Roads and Bridges », Ford Foundation	🏛️ Primaire	2016	EN
22	N. Eghbal, « Working in Public », Stripe Press	🏛️ Primaire	2020	EN
23	TechCrunch, « Open source sustainability »	📰 Presse spéc.	06/2018	EN
24	The Register, « When software depends on a random guy in Nebraska »	📰 Presse spéc.	05/2021	EN
25	OpenSSF, « SBOM Standards: EU CRA and OpenSSF »	🏛️ Primaire	10/2025	EN

#	Source	Type	Date	Langue
26	Sonatype, « EU CRA: Good for Software Supply Chain Security, Bad for Open Source? »	📰 Presse spéc.	12/2022	EN
27	NCSC UK, « SBOMs and the importance of inventory »	🏛️ Primaire	03/2025	EN
28	Commission européenne, « Cyber Resilience Act »	🏛️ Primaire	2024	EN
29	FOSSA, « SBOM Requirements in the EU's CRA »	📰 Presse spéc.	09/2024	EN

Recherche effectuée en mars 2026. Langues : français, anglais. Sources sur le Sud global issues de la session 1 (RAFLL, communautés francophones du libre). Toutes les URL visitées pendant la recherche.

← L'enclosure du code — Série complète

floss - 3 - L'enclosure du code: l'ère générative

2026 / 04 / 19 · https://hyper.dev/2026/enclosure-du-code/3-ere-generative/

Comment l'IA a achevé l'arc d'extraction — en transformant les communs logiciels en matière première pour commodifier le travail même qui les avait construits

Troisième session — La chaîne d'approvisionnement logicielle, de 2021 à aujourd'hui

← L'enclosure du code — Série complète

---

1. Synthèse

[INTERPRÉTATION] L'IA générative représente l'achèvement d'un arc d'extraction de trois décennies. L'ère 1 (1990-2010) a recadré le vocabulaire de la liberté en vocabulaire d'efficacité, rendant l'extraction culturellement acceptable. L'ère 2 (2010-2022) a normalisé la construction d'empires commerciaux sur des communs non financés. L'ère 3 (2022-présent) a franchi le dernier pas : le code produit dans les communs est devenu la matière première d'entraînement pour des outils qui commodifient l'acte même d'écrire du code — rendant potentiellement obsolètes les personnes dont le travail a nourri le modèle. 🟡 70-89%

Mais cette lecture, aussi structurellement solide soit-elle, n'est pas la seule. L'ère générative a *aussi* représenté la démocratisation la plus radicale de la capacité à construire du logiciel que le monde ait connue. Un lycéen à Cotonou ou Tizi Ouzou peut désormais faire en une après-midi ce qui

nécessitait une équipe d'ingénieurs il y a dix ans. La question n'est pas « est-ce bon ou mauvais ? » mais « pour qui ? » — et les premières données suggèrent que la réponse est brutalement inégale. L'emploi des développeurs de 22-25 ans aux États-Unis a chuté de près de 20% entre 2022 et 2025. Parallèlement, 40% des tâches dans le secteur d'externalisation technologique africain pourraient être affectées par l'IA d'ici 2030. La redistribution est en cours — mais elle n'est ni universelle ni équitable.

---

2. Résultats détaillés

2.1 Les communs comme données d'entraînement

Le procès Copilot : le code comme bien commun ou comme propriété ?

[FAIT] Le 3 novembre 2022, le cabinet Joseph Saveri Law Firm et l'avocat-programmeur Matthew Butterick ont déposé une plainte collective contre GitHub, Microsoft et OpenAI devant le tribunal fédéral de Californie du Nord. L'action allègue que GitHub Copilot — entraîné sur des milliards de lignes de code publiquement disponible sur GitHub — viole les conditions des licences open source en générant du code dérivé sans attribution, sans notice de copyright et sans termes de licence. L'action demande des dommages potentiellement supérieurs à 9 milliards de dollars. 🟢 90-100%

[FAIT] GitHub admet dans sa FAQ que Copilot reproduit parfois des extraits de code de plus de 150 caractères identiques aux données d'entraînement — environ 1% du temps. Une analyse indépendante a trouvé que dans les fichiers où Copilot est activé, l'IA produit jusqu'à 40% du code en Python. En 2026, le procès est toujours en cours. Le tribunal a rejeté les prétentions de violation du DMCA (les extraits de Copilot

n'étant pas suffisamment identiques au code original), mais a maintenu les prétentions de rupture de contrat et de violation des licences open source. Le fond de l'affaire n'est pas tranché. 🟢 90-100%

[INTERPRÉTATION] La question centrale est précise : les licences open source existantes — GPL, MIT, Apache — ont-elles jamais anticipé qu'une entreprise pourrait ingérer l'intégralité du corpus de code sous licence, l'utiliser pour entraîner un modèle commercial, et revendre le résultat sans respecter aucune des conditions de licence ? La réponse est presque certainement non. Ces licences ont été écrites pour un monde où le code était copié, modifié et redistribué par des humains. Le modèle d'entraînement de l'IA crée une forme d'utilisation qui n'existait pas conceptuellement quand les licences ont été rédigées. 🟡 70-89%

Les deux arguments

L'argument de la « marée montante » : L'IA rend tout le monde plus productif. Copilot aide les développeurs à écrire du code plus vite, réduit le temps passé sur le boilerplate, et permet de se concentrer sur la logique de haut niveau. GitHub estime que la productivité améliorée par les assistants IA pourrait ajouter plus de 1 500 milliards de dollars au PIB mondial.

L'argument de l'enclosure : C'est l'extraction finale des communs. Des décennies de travail bénévole et communautaire — le code ouvert qui fait fonctionner Internet — ont été ingérées par des entreprises privées pour créer des outils qui concurrencent les personnes mêmes dont le travail a été utilisé. Comme l'a écrit Azer Koçulu en 2016 lors de l'incident left-pad : « npm est le terrain privé de quelqu'un, où les entreprises sont plus puissantes que les personnes. »

Remplacez « npm » par « GitHub » et « Copilot » : la structure est identique, mais l'échelle est transformée.

2.2 Déqualification ou démocratisation : ce que les données montrent

Les données qui dérangent

[FAIT] Une étude de Stanford (Digital Economy Study) a trouvé que l'emploi des développeurs de 22-25 ans aux États-Unis a chuté de près de 20% entre fin 2022 et mi-2025 — coïncidant avec l'essor des outils de codage IA. Les offres d'emploi pour développeurs ont chuté d'environ 70% depuis leur pic de 2022 selon Indeed. Les stages en technologie ont diminué de 30% depuis 2023 selon Handshake. Salesforce a annoncé l'arrêt des embauches de développeurs juniors en 2025, citant les gains de productivité IA. 🟡 70-89%

[FAIT] L'enquête Stack Overflow 2025 (la plus récente disponible) montre que 84% des développeurs utilisent des outils IA dans leur flux de travail — contre 70% en 2024. Mais le sentiment positif envers ces outils a *diminué* : de plus de 70% en 2023-2024 à seulement 60% en 2025. Plus révélateur encore : 46% des développeurs méfient *activement* de la précision des outils IA, contre seulement 33% qui leur font confiance. Les développeurs expérimentés sont les plus méfiants. 🟢 90-100%

[FAIT] L'étude METR (2025), un essai contrôlé randomisé avec 16 développeurs expérimentés réalisant 246 tâches dans leurs propres projets open source, a produit un résultat surprenant : quand les outils IA étaient autorisés, les développeurs prenaient 19% *plus longtemps* à compléter les tâches — l'IA les ralentissait. Les développeurs eux-mêmes estimaient que l'IA

les avait accélérés de 20%. L'écart entre perception et réalité est de 39 points de pourcentage. 🟡 70-89%

[FAIT] Le rapport Google DORA 2024 a trouvé que l'utilisation accrue de l'IA améliore la vitesse de documentation mais cause une baisse de 7,2% de la stabilité de livraison. Le codage assisté par IA produit quatre fois plus de clonage de code, augmentant la dette technique. Le « code churn » — le pourcentage de code jeté moins de deux semaines après avoir été écrit — devait doubler en 2024. 🟡 70-89%

L'atrophie des compétences

[INTERPRÉTATION] Le cas de Luciano Nooijen, ingénieur chez Companion Group, est emblématique : après avoir utilisé intensivement les outils IA au travail, il s'est retrouvé en difficulté lors d'un projet personnel sans accès à ces outils. « Je me sentais stupide parce que des choses qui étaient instinctives sont devenues manuelles, parfois même pénibles », a-t-il déclaré au MIT Technology Review. Sa conclusion : comme les athlètes qui s'entraînent encore aux gestes de base, la seule façon de maintenir l'instinct du code est de pratiquer régulièrement le travail de fond. 🟡 70-89%

Le concept de « vibe coding » — inventé par Andrej Karpathy (OpenAI/Tesla) en février 2025, désignant l'approche où l'on décrit le logiciel en langage naturel et on laisse l'IA écrire le code — cristallise le risque de déqualification. Les critiques parlent d'une « dette technique archéologique » : du code qui fonctionne aujourd'hui mais que personne ne comprend, dont l'historique des commits dit « améliorations IA » sans expliquer la logique, et que les développeurs futurs devront rétro-ingéniérer comme des anthropologues étudiant une civilisation ancienne à travers ses artefacts.

2.3 Le « superviseur IA » : restructuration de la profession

[FAIT] Le secteur technologique mondial a éliminé environ 244 851 emplois en 2025 selon RationalFX. L'IA et l'automatisation figurent parmi les facteurs les plus fréquemment cités. Microsoft a licencié 6 500 personnes, son CEO déclarant que les outils IA écrivent désormais 20-30% du nouveau code de l'entreprise. IBM a remplacé ~8 000 postes RH par un chatbot IA. Accenture a supprimé plus de 11 000 postes en trois mois, la CEO Julie Sweet déclarant que le recyclage « n'est pas un chemin viable » pour les compétences nécessaires. Fiverr a réduit ses effectifs de 30%, se repositionnant comme « AI-first ». 🟢 90-100%

[INTERPRÉTATION] Le schéma est structurel, pas conjoncturel. Les entreprises affichent des profits records tout en citant les « gains d'efficacité IA » pour justifier des réductions d'effectifs. Amazon a annoncé fin 2025 la suppression de jusqu'à 30 000 postes corporate, alors que son chiffre d'affaires trimestriel était en hausse de 11%. La déconnexion entre performance d'entreprise et sécurité des travailleurs n'a jamais été aussi nette. Les gains de productivité vont aux actionnaires ; les travailleurs qui restent sont priés de faire plus avec des outils IA, pour la même rémunération ou moins. 🟡 70-89%

Le discours managérial oscille entre deux pôles. Le CEO de Google, Sundar Pichai, affirme que l'IA « rend les ingénieurs plus productifs en éliminant les tâches fastidieuses ». Le CEO d'Amazon Web Services, Matt Garman, insiste sur la nécessité de continuer à embaucher et former des juniors « autant que jamais ». Mais le CEO de Salesforce, Marc Benioff, a gelé les embauches d'ingénieurs en 2025. Les actes contredisent souvent les paroles.

2.4 Qui cela frappe le plus durement : l'asymétrie mondiale

[FAIT] 40% des tâches dans le secteur d'externalisation technologique africain en croissance pourraient être affectées par l'IA d'ici 2030 selon un rapport Mastercard Foundation/Caribou. Les rôles d'expérience client, qui représentent 44% de l'emploi BPO, sont particulièrement vulnérables — la moitié des tâches pourraient être automatisées. Seules 10% des tâches du secteur sont pleinement résilientes à l'automatisation. 🟡 70-89%

[FAIT] Rest of World a documenté en 2025 la chaîne d'approvisionnement cachée du travail IA en Afrique : des travailleurs dans 39 nations africaines sont employés par des sous-traitants situés aux Émirats, en Amérique du Nord et en Europe, fournissant des données à des clients comme Meta, OpenAI et Samsung. Une enquête Equidem de 2025 auprès de 76 travailleurs en Colombie, au Ghana et au Kenya a rapporté 60 incidents indépendants de dommages psychologiques — anxiété, dépression, PTSD. Le sociologue Antonio Casilli (Institut Polytechnique de Paris) décrit un « cercle d'invisibilité autour de ce travail ». 🟢 90-100%

[FAIT] L'ONU note qu'entre 2020 et 2024, dix pays ont concentré près de 80% de tous les investissements numériques « greenfield » dans le Sud global — aucun n'était en Afrique. La pénétration internet en Afrique varie de 12,5% (Burundi) à 92,6% (Maroc). Plus des deux tiers de la capacité de centres de données du continent sont concentrés en Afrique du Sud. 🟡 70-89%

[INTERPRÉTATION] Le paradoxe de l'ère générative pour le Sud global est complet. D'un côté, les outils IA sont accessibles — un développeur à Nairobi utilise le même Copilot qu'un

développeur à San Francisco. De l'autre, ces outils menacent exactement les filières d'emploi que les ères précédentes avaient créées : l'externalisation de tâches de codage, de modération de contenu, de traitement de données. L'Afrique fournit le travail invisible qui entraîne les modèles (annotation, modération, étiquetage) et subit les conséquences de l'automatisation des tâches que ce même travail a rendu possible. C'est une double extraction : extraction du travail d'entraînement, puis remplacement des travailleurs par les systèmes que leur travail a entraînés.

2.5 Les enjeux humains au bout de l'arc

L'identité en chute libre

[INTERPRÉTATION] Que signifie « être développeur » quand la machine produit du code « suffisamment bon » ? L'enquête Stack Overflow 2025 montre que pour la première fois, les développeurs seniors rapportent une satisfaction professionnelle plus basse que les juniors. Le burnout n'est plus subtil — il est bruyant. Le développeur moderne ne code plus seulement : il gère des outils, des fuseaux horaires, des scripts Terraform et du burnout. L'identité artisanale de l'ère 1 — « je comprends mes outils, donc je suis » — a été érodée par l'assemblage de l'ère 2, puis achevée par l'ère 3 où même l'assemblage peut être délégué à un modèle qu'on ne peut pas inspecter.

L'analogie avec le journalisme est la plus parlante pour un public non technique : imaginez un siècle d'archives journalistiques en ligne, utilisé pour entraîner un système qui génère des articles « suffisamment bons » pour une fraction du coût. Les journalistes dont le travail a entraîné le système sont

désormais en concurrence avec lui. C'est approximativement ce qui arrive aux développeurs.

La réponse politique : toujours en retard d'une ère

[FAIT] L'EU AI Act, entré en vigueur progressivement depuis 2024, inclut des règles sur l'exploration de textes et de données (text and data mining). L'article 53 impose aux fournisseurs de modèles d'IA à usage général de publier des résumés « suffisamment détaillés » de leurs données d'entraînement. Le US Copyright Office n'a pas encore tranché la question de la protégeabilité du code généré par IA. Le Royaume-Uni explore des changements législatifs pour équilibrer les besoins de recherche avec les droits des créateurs. 🟡 70-89%

[INTERPRÉTATION] La politique est structurellement en retard. L'Executive Order de 2021 et le CRA européen répondent à des incidents passés (SolarWinds, Log4Shell). L'AI Act commence à adresser les modèles génératifs, mais les questions fondamentales — ce qui est dû aux auteurs du code sur lequel les modèles sont entraînés, le statut juridique du code généré par IA, la responsabilité quand du code IA vulnérable entre en production — restent sans réponse claire dans la plupart des juridictions. Les organismes non occidentaux (Union Africaine, ASEAN) en sont aux cadres stratégiques (la Stratégie de Transformation Numérique de l'UA 2020-2030), loin des régulations opérationnelles.

L'agentivité au bout de l'arc : de la maîtrise au prompt

Voici la trajectoire complète, résumée :

Ère 1 (1990-2010) : le développeur possède et comprend ses outils. L'identité professionnelle repose sur la maîtrise. Le

mouvement du logiciel libre donne à cette maîtrise une expression juridique et communautaire. Le recadrage « open source » détache la maîtrise de la liberté, mais le développeur comprend encore ce qu'il fait.

Ère 2 (2010-2022) : le développeur assemble des paquets qu'il ne contrôle pas entièrement. Le graphe de dépendances est ingouvernable. L'identité se déplace de la maîtrise vers la productivité : ce qui compte, c'est la vitesse de livraison, pas la compréhension profonde. La communauté se fragmente en communautés d'utilisateurs de frameworks plutôt qu'en communautés de pratique.

Ère 3 (2022-présent) : le développeur prompte un modèle qu'il ne peut pas inspecter, entraîné sur du travail qui n'a pas été rémunéré. La productivité individuelle peut augmenter, mais l'agentivité — la capacité à comprendre, modifier et posséder ses propres outils — diminue. Le sentiment de compétence devient découplé du sentiment de compréhension : on peut produire du code qui fonctionne sans savoir pourquoi il fonctionne.

---

3. Contradictions et signaux faibles

L'IA ralentit les développeurs expérimentés

Le résultat le plus dérangeant de cette recherche est l'étude METR : dans un essai contrôlé randomisé, les développeurs expérimentés avec outils IA étaient 19% *plus lents* que sans — tout en croyant être 20% plus rapides. L'écart de 39 points suggère que les rapports de productivité massive des entreprises vendeuses d'outils IA (GitHub, Google, Microsoft) pourraient être fondamentalement biaisés. C'est un seul

résultat, sur 16 développeurs, et il mérite d'être répliqué. Mais il suffit à introduire un doute sérieux sur le récit dominant.

La redistribution brutale plutôt que la perte universelle

La thèse de l'extraction totale — « l'IA prend tout et ne rend rien » — est trop simple. Les données suggèrent plutôt une redistribution brutale : certaines personnes gagnent en agentivité (le développeur solo qui utilise l'IA comme amplificateur pour expédier un produit), tandis que d'autres la perdent (le développeur junior dont le poste est supprimé, le mainteneur dont le code a entraîné le modèle sans compensation, le travailleur africain qui annotait les données et qui est maintenant remplacé par le système que son travail a rendu possible). Le gain des uns est structurellement lié à la perte des autres.

Le sentiment de compétence sans la compétence

Le « vibe coding » crée un phénomène nouveau : des personnes capables de produire du logiciel fonctionnel sans comprendre ce qu'elles ont produit. Ceci est simultanément libérateur (pour la personne qui peut enfin réaliser une idée) et dangereux (pour la société qui dépend de la fiabilité de ce logiciel). La question n'est pas « est-ce que ça marche ? » mais « qui le répare quand ça casse ? » — et si la réponse est « personne, parce que personne ne comprend le code », alors nous avons un problème de société, pas seulement un problème technique.

---

4. Angles morts

Données longitudinales sur l'emploi des développeurs. L'étude Stanford sur la chute de 20% de l'emploi des 22-25 ans est un

signal fort, mais la corrélation avec l'IA n'est pas une causalité démontrée. Des facteurs conjoncturels (fin du boom post-COVID, hausse des taux d'intérêt, récession du VC) contribuent aussi. Des études longitudinales avec contrôle des variables confondantes n'existent pas encore.

L'expérience vécue des mainteneurs dont le code a entraîné les modèles. Le procès Copilot documente le cadre juridique, mais les voix individuelles des développeurs qui ont découvert leur code dans les sorties de Copilot sont sous-documentées dans les sources accessibles.

L'impact sur les communautés open source. Les taux de contribution open source depuis le lancement de Copilot n'ont pas été systématiquement mesurés dans les sources trouvées. La question de savoir si l'IA remplace la collaboration communautaire par le travail solo assisté par IA est cruciale — et les données manquent.

La perspective des éducateurs en informatique. Comment les programmes de CS adaptent-ils (ou non) leurs curricula à l'ère IA ? Des études qualitatives sur les pratiques d'enseignement seraient essentielles.

Le cadre juridique hors US/EU. Les positions de l'Union Africaine, de l'ASEAN, du Mercosur et d'autres organismes non occidentaux sur le droit d'auteur du code IA sont pratiquement absentes des sources trouvées.

L'effet sur la sécurité du logiciel à moyen terme. Si 48% du code généré par IA contient des vulnérabilités potentielles (comme le suggèrent certaines études), et si ce code entre massivement en production, les conséquences sur la sécurité pourraient être considérables — mais nous n'avons pas encore les données pour le mesurer.

5. Synthèse actionnable

L'arc complet : ce que les trois ères révèlent

Si l'on devait expliquer cette trajectoire à quelqu'un qui n'a jamais écrit une ligne de code, dans un langage qu'il comprend déjà, voici ce qu'on dirait :

L'analogie de l'enclosure agricole. Au XVIIIe siècle en Angleterre, les terres communes — utilisées par les villageois pour paître leurs animaux — ont été progressivement clôturées par des propriétaires privés. La productivité agricole a augmenté. Les villages ont été vidés. Les anciens commoners sont devenus des ouvriers dans les usines de ceux qui avaient clôturé leurs terres. L'ère 1 a supprimé les clôtures conceptuelles (le logiciel libre a créé les communs numériques), puis les a remplacées par des clôtures plus subtiles (le recadrage « open source »). L'ère 2 a construit des châteaux sur les communs (AWS, GitHub, les plateformes cloud). L'ère 3 a moissonné les communs eux-mêmes comme matière première pour construire des machines qui remplacent les commoners.

L'analogie de la consolidation médiatique. Des décennies de journalisme archivé en ligne ont été utilisées pour entraîner des systèmes qui produisent des articles « suffisamment bons » pour une fraction du coût. Les journalistes dont le travail a entraîné le système sont en concurrence avec lui. Les plateformes qui distribuent le contenu (Google, Facebook) captent la valeur publicitaire. Le producteur de contenu original est doublement dépossédé : son travail a nourri le modèle, et le modèle le remplace sur le marché.

L'analogie de la grande distribution. Le développeur open source est le petit producteur. La plateforme cloud est l'hypermarché. L'IA générative est la marque distributeur : le produit est fabriqué à partir des mêmes ingrédients que le produit original, mais il porte le logo de l'hypermarché et se vend moins cher.

Ce qui peut être fait

Les leviers d'action existent, mais ils sont distribués et aucun n'est suffisant seul :

Le levier juridique. Le procès Copilot, s'il aboutit à une reconnaissance que l'entraînement sur du code sous licence viole les conditions de licence, créerait un précédent fondamental. La question de savoir si les licences open source existantes couvrent l'entraînement de modèles est le champ de bataille juridique de la décennie.

Le levier des licences. De nouvelles licences ou amendements explicitement interdisant l'entraînement IA sans consentement commencent à émerger. Certains développeurs ajoutent des restrictions « no AI training » dans leurs fichiers LICENSE. Ces approches sont imparfaites mais signalent une prise de conscience.

Le levier politique. Le CRA européen et l'AI Act sont des premiers pas. Mais une réponse adéquate exigerait de reconnaître que la chaîne d'approvisionnement logicielle est une infrastructure critique — et que les mainteneurs qui la soutiennent méritent un soutien structurel, pas seulement des obligations de conformité.

Le levier communautaire. Les communautés du logiciel libre — des RAFLL en Afrique francophone aux fondations Apache

et Linux — restent les seuls espaces où la question « à qui appartient ce que les communs produisent ? » peut être posée sans être immédiatement récupérée par un intérêt commercial. Leur survie et leur vitalité sont un enjeu stratégique, pas seulement culturel.

Le levier individuel. Le principe de Smalltalk — « si un système doit servir l'esprit créatif, il doit être entièrement compréhensible par un seul individu » — reste la meilleure boussole. Les développeurs qui maintiennent leur capacité à comprendre leurs outils de bout en bout, qui refusent la déqualification passive, qui pratiquent le « codage de fond » comme un athlète pratique les gammes, sont ceux qui conservent leur agentivité. Ce n'est pas un conseil de développement personnel — c'est un acte de résistance structurelle.

---

6. Pistes d'approfondissement

Études longitudinales sur l'emploi. Des études avec contrôle des variables confondantes sont nécessaires pour distinguer l'impact de l'IA de l'impact des cycles économiques sur l'emploi des développeurs.

Réplication de l'étude METR. Le résultat « l'IA ralentit les développeurs expérimentés de 19% » est potentiellement le résultat le plus important de cette recherche — et le plus sous-couvert médiatiquement. Il doit être répliqué à plus grande échelle.

Impact sur les contributions open source. Mesurer systématiquement les taux de contribution, la diversité des contributeurs, et la santé des communautés open source depuis le lancement de Copilot.

Cartographie du travail invisible. Documenter les chaînes d'approvisionnement du travail IA — annotation, modération, étiquetage — avec une attention particulière aux travailleurs du Sud global, en s'appuyant sur le travail de l'African Content Moderators Union, de Personaldata.io et du Distributed AI Research Institute.

Cadre juridique francophone. Explorer les positions spécifiques de la France (ANSSI, CNIL), de la Belgique, du Canada et des pays d'Afrique francophone sur le droit d'auteur du code généré par IA et les implications du CRA pour les communautés open source.

---

7. L'arc complet : ce que ces trois ères ont révélé

Trois décennies. Trois ères. Un seul arc.

L'ère 1 a créé les communs numériques et les a immédiatement recadrés dans un vocabulaire qui rendait l'extraction acceptable. Le mouvement du logiciel libre a dit : « la liberté est le but ». Le mouvement open source a répondu : « l'efficacité est le but ». Les deux ont produit du code extraordinaire. Mais le second a ouvert les portes que les suivants ont traversées.

L'ère 2 a construit des empires au-dessus des communs. Les plateformes cloud ont encapsulé l'infrastructure. Les gestionnaires de paquets ont encapsulé les dépendances. Le développeur est passé d'artisan à assembleur. L'extraction — un fournisseur cloud qui forke un projet open source et le revend comme service — est devenue le fonctionnement normal du marché. Les mainteneurs qui tenaient l'infrastructure debout étaient invisibles, non payés, et en burnout.

L'ère 3 a moissonné les communs eux-mêmes. Le corpus GitHub — 200 millions de dépôts, des décennies de travail collectif — est devenu données d'entraînement. Les outils résultants commodifient l'acte d'écrire du code, menaçant exactement les personnes dont le travail a nourri le modèle. Les développeurs juniors voient leurs perspectives d'emploi s'effondrer. Les travailleurs du Sud global, qui annotaient les données et maintenaient les services externalisés, sont menacés par l'automatisation des tâches que leur travail a rendu possible.

Le schéma est celui de toutes les enclosures : des communs productifs, appropriés par des acteurs privés qui en extraient la valeur tout en externalisant les coûts sur ceux qui les avaient construits. La particularité ici est que l'extraction finale ne prend pas seulement le produit (le code) mais le savoir-faire (la capacité d'écrire du code) — en le transformant en poids statistiques dans un modèle que personne ne peut inspecter.

Ceci n'est pas une fatalité. Les alternatives existent : des licences qui anticipent l'IA, des communautés qui maintiennent l'exigence de compréhension, des politiques qui reconnaissent les mainteneurs comme infrastructure critique, des systèmes compréhensibles par un seul individu. Mais ces alternatives demandent un travail délibéré, soutenu, et collectif. Elles ne se produiront pas par défaut.

Le défaut, c'est l'enclosure.

> *« Non pas que tout le monde pense la même chose, mais que la pensée de chacun laisse une trace. »*

---

8. Table des sources

#	Source	Type	Date	Langue
1	Joseph Saveri Law Firm, page du procès Copilot	🏛️ Primaire	2022-2026	EN
2	Yang Law Offices, « GitHub Copilot Lawsuit: What Developers Need to Know »	📰 Presse spéc.	12/2025	EN
3	Finnegan LLP, « Insights from the Pending Copilot Class Action Lawsuit »	📰 Presse spéc.	2024	EN
4	The Register, « GitHub Copilot copyright case narrowed but not neutered »	📰 Presse spéc.	01/2024	EN
5	NYU JILP, « Do As I Say, Not As I Code: GitHub's Copilot Prompts IP Litigation »	🎓 Académique	03/2025	EN
6	PatentAILab, « GitHub Copilot Lawsuit Update (Feb 2026) »	📰 Presse spéc.	02/2026	EN
7	METR, « Measuring the Impact of Early-2025 AI on Experienced OS Developer Productivity »	🎓 Académique	07/2025	EN
8	MIT Technology Review, « AI coding is now everywhere. But not everyone is convinced »	📰 Presse gén.	12/2025	EN
9	Stack Overflow, « AI vs Gen Z: How AI has changed the career pathway for junior developers »	🏛️ Primaire	12/2025	EN

#	Source	Type	Date	Langue
10	Stack Overflow, 2025 Developer Survey (section AI)	🏛️ Primaire	2025	EN
11	CodeConductor, « Junior Developers in the Age of AI (2026 Guide) »	📰 Presse spéc.	01/2026	EN
12	DevOps.com, « AI in Software Development: Productivity at the Cost of Code Quality? »	📰 Presse spéc.	12/2025	EN
13	Second Talent, « AI Coding Assistant Statistics & Trends [2025] »	📰 Presse spéc.	10/2025	EN
14	Crunchbase, « Tech Layoffs: US Companies With Job Cuts In 2024 And 2025 »	📰 Presse spéc.	03/2026	EN
15	AllAboutAI, « Tech Layoffs Hit 157,000 in 2025 and AI Is Both the Cause and the Excuse »	📰 Presse spéc.	11/2025	EN
16	Network World, « Global tech-sector layoffs surpass 244,000 in 2025 »	📰 Presse spéc.	01/2026	EN
17	Programs.com, « List of Companies Announcing AI-Driven Layoffs »	📰 Presse spéc.	12/2025	EN
18	FinalRoundAI, « How AI Vibe Coding Is Destroying Junior Developers' Careers »	📝 Blog	2025	EN

#	Source	Type	Date	Langue
19	Mastercard Foundation / Caribou, « 40% Of Tasks In Africa's Tech Outsourcing May Be Affected by AI by 2030 »	🏛️ Primaire	04/2025	EN
20	Rest of World, « How Big Tech's AI labor supply chain relies on hidden African workers »	📰 Presse gén.	05/2025	EN
21	Brookings, « Reimagining the future of data and AI labor in the Global South »	🎓 Académique	10/2025	EN
22	UNU, « AI and Africa's Future Of Work — Mozambique's Moment Of Decision »	🏛️ Primaire	12/2025	EN
23	AfriCatalyst, « AI is Rewriting the Future of Work. What Does That Mean for Africa? »	📰 Presse spéc.	08/2025	EN
24	Commission européenne, « Cyber Resilience Act »	🏛️ Primaire	2024	EN
25	CISA, « Apache Log4j Vulnerability Guidance »	🏛️ Primaire	12/2021	EN

Recherche effectuée en mars 2026. Langues : français, anglais. Certaines sources africaines issues des sessions précédentes (RAFLL, communautés francophones du libre) sont référencées dans le texte sans être reprises dans la table. Toutes les URL visitées pendant la recherche.

← L’enclosure du code — Série complète

« Avant les plateformes », « L’ère de l’assemblage », « L’ère générative » — composent un arc complet, de la naissance du logiciel libre aux conséquences de l’IA générative sur les personnes qui écrivent du code. L’arc peut être lu dans l’ordre ou dans le désordre; chaque document se tient seul, mais les trois ensemble racontent une histoire que les parties séparées ne racontent pas.

floss - 4 - L'enclosure du code: le prélude et l'après

2026 / 04 / 19 · https://hyper.dev/2026/enclosure-du-code/fin-le-prelude-et-apres/

Ce que le crépuscule a à dire que l'aube a besoin d'entendre

Prologue et épilogue de la trilogie — La chaîne d'approvisionnement logicielle

← L'enclosure du code — Série complète

---

0. Pourquoi ce texte existe

Les trois sessions précédentes ont tracé un arc : de la liberté recadrée (ère 1), aux communs encapsulés (ère 2), aux communs moissonnés (ère 3). Mais cet arc a un avant et un après.

L'avant, c'est le prélude — les pionniers d'avant GNU, ceux pour qui le logiciel n'était pas encore un bien à posséder ni un commun à défendre, parce que la question ne se posait pas. Le temps où le code était un geste partagé comme on partage une recette de cuisine.

L'après, c'est la question que l'arc pose sans y répondre : maintenant qu'on sait, qu'est-ce qu'on fait ?

Ce texte n'est pas une recherche. C'est une conversation entre deux voix : le crépuscule (celui qui a traversé les trois ères, qui a des cicatrices et des outils) et l'aube (celui qui arrive dans un paysage dont il n'a pas choisi la forme). Le crépuscule n'a pas raison. L'aube n'est pas naïve. Ils ont besoin l'un de l'autre.

I. Le prélude : avant que la question ne se pose

Le crépuscule

Avant GNU, avant la FSF, avant que quiconque ait besoin de défendre le droit de lire le code source, il y avait une époque où la question n'existait pas. Dans les laboratoires du MIT, de Bell Labs, de Xerox PARC dans les années 1960-1970, le logiciel circulait comme les articles scientifiques circulaient : on le donnait parce que c'est ce qu'on faisait. Le code était un sous-produit du matériel, pas une marchandise. Personne ne vendait du logiciel parce que personne n'imaginait que c'était vendable.

Ce qui circulait dans ces laboratoires, c'était quelque chose de plus précis que du « code ». C'étaient des idées incarnées — des façons de penser rendues exécutables. Quand les gens de Xerox PARC ont inventé Smalltalk, ils n'inventaient pas un langage de programmation. Ils inventaient une philosophie : que la complexité doit être compréhensible, qu'un système doit être entièrement saisissable par un seul esprit humain, que l'interface est le système. Quand les gens du MIT ont créé Lisp, ils ne créaient pas un outil — ils formalisaient une manière de raisonner où le programme et les données sont la même chose, où la structure est transparente, où rien n'est caché.

L'aube

Mais je n'ai pas accès à ça. Quand je démarre, je reçois un framework de 41 000 fichiers, un tutoriel de 20 minutes, et un Copilot qui m'autocomplète les lignes. Je n'ai jamais vu la question se poser parce qu'elle a été résolue avant moi — résolue dans un sens qui m'est défavorable. Le code circule, oui, mais sur les conditions de quelqu'un d'autre. La

philosophie du PARC, je la lis dans des livres. Le laboratoire n'existe plus.

Le crépuscule

C'est vrai. Et c'est le point. Le prélude n'est pas un âge d'or. Les laboratoires du PARC et du MIT étaient des espaces de privilège extrême — presque exclusivement blancs, masculins, anglophones, financés par le département de la Défense ou par des entreprises qui pouvaient se permettre de la recherche pure. La communauté hacker des années 1970 était une communauté de quelques centaines de personnes dans quelques institutions américaines. Le partage était réel — mais il existait dans un contexte si restreint que la question de la propriété ne se posait pas, non pas parce qu'il y avait une éthique universelle du partage, mais parce qu'il n'y avait pas assez de monde pour que le conflit émerge.

Le moment où le logiciel est devenu une marchandise — la lettre de Bill Gates aux hobbyistes en 1976, la fermeture du code source par AT&T pour Unix, la montée de Microsoft dans les années 1980 — n'était pas une corruption d'un état naturel. C'était le résultat prévisible de l'industrialisation. Quand le marché arrive, la question de la propriété arrive avec lui.

Ce que Stallman a fait en 1983, en lançant GNU, c'était de refuser de traiter cette industrialisation comme naturelle. Il a dit : non, nous pouvons organiser le logiciel autrement. Il avait raison. Mais l'arc que nous avons tracé montre ce qui arrive quand une alternative existe mais que le système l'absorbe.

L'aube

Alors le prélude n'est pas un modèle à restaurer ?

Le crépuscule

Non. C'est un rappel que les choses n'ont pas toujours été ainsi. Ce qui veut dire qu'elles n'ont pas à rester ainsi. Le prélude montre que la possession du code est un choix civilisationnel, pas une loi de la physique. Un autre choix a été fait — et peut être refait.

II. Ce que le crépuscule voit en 2026

Le crépuscule

Voici ce que je vois après trente ans dans le métier, dit honnêtement.

Le tenure ne protège plus. Il y a cinq ans, vingt ans d'expérience étaient un bouclier. Aujourd'hui, vingt ans d'expérience signifient vingt ans de dette technique que tu portes dans ta tête — des frameworks disparus, des patterns obsolètes, des intuitions formées dans un monde qui n'existe plus. L'expérience qui compte est celle qui se reconfigure. L'expérience qui ne se reconfigure pas est un passif.

Le CV-driven development est un piège mortel. J'ai vu des collègues empiler Kubernetes, des ORM, React, TypeScript, GraphQL, Terraform — non pas parce que le problème l'exigeait, mais parce que leur prochain entretien d'embauche l'exigeait. Ils ont optimisé leur employabilité au prix de leur compréhension. Quand le modèle de langage sait configurer Kubernetes aussi bien qu'eux, il ne leur reste rien. Ils ont appris les commandes, pas les principes. La surface, pas la structure.

La simplification est le vrai luxe. Dans un monde noyé sous la complexité, la capacité à supprimer du code, à réduire les dépendances, à dire « nous n'avons pas besoin de ça » — cette capacité vaut plus que toute la pile technique du monde. Mais elle n'est pas valorisée. Personne n'a jamais été promu pour avoir supprimé un microservice. Personne n'a jamais été embauché parce que son système n'avait que trois dépendances. L'industrie récompense la complexité parce que la complexité justifie des effectifs, des budgets, et des consultants. La simplicité menace tout ça.

Mais attention : il y a deux figures derrière « je comprends en profondeur », et elles sont opposées. La première regarde un système de 47 microservices et dit : « ceci est un monolithe déguisé, remplaçons-le par un service et une base de données ». Cette personne crée de la valeur en *réduisant* la complexité. La seconde est le rentier du brownfield — celui dont le pouvoir vient exclusivement du fait qu'il est le seul à comprendre pourquoi l'ORM fait ce truc bizarre le mardi, pourquoi le pipeline CI a ce hack que personne ne touche, pourquoi le déploiement nécessite trois scripts shell non documentés dans un ordre précis. Cette personne extrait de la rente d'une complexité qu'elle a contribué à créer ou qu'elle a laissé s'accumuler. Son « expertise profonde » n'est pas de la maîtrise — c'est du lock-in personnel.

Le test est simple : est-ce que ta contribution rend le système *plus compréhensible par quelqu'un d'autre que toi*, ou *moins* ? Si la réponse est « moins », tu n'es pas un artisan. Tu es un péage.

L'aube

Mais comment je survis, concrètement ? Les offres d'emploi demandent Kubernetes. Les bootcamps enseignent React. Les managers veulent du « full-stack » avec 47 technologies dans la pile. Si je simplifie, je ne suis pas embauché.

Le crépuscule

Tu as raison sur le marché tel qu'il est. Mais le marché tel qu'il est — celui qui embauche 70% de moins qu'en 2022, qui licencie 244 000 personnes en 2025, qui remplace les juniors par des outils IA — ce marché ne te veut pas non plus avec 47 technologies. Ce marché est en train de se réorganiser. Et quand il aura fini, il aura besoin de personnes qui comprennent les *problèmes*, pas les *outils*.

Voici ma prédiction, aussi réaliste que je peux la faire.

III. Prédictions honnêtes

Ce qui va probablement arriver

1. Le SaaS actuel va s'effondrer partiellement. Le modèle SaaS de la dernière décennie — des applications identiques vendues à des milliers d'entreprises avec une personnalisation minimale — est menacé des deux côtés. Par le haut : l'IA permet de générer des applications « suffisamment bonnes » sur mesure, rendant les SaaS génériques moins nécessaires. Par le bas : les coûts d'infrastructure cloud deviennent insoutenables pour les entreprises moyennes. Ce qui survivra, c'est le SaaS qui résout un problème de domaine si précis qu'aucun outil général ne peut le remplacer. Le SaaS générique — le énième outil de gestion de projet, le énième CRM — sera le premier à tomber.

2. Le « logiciel situé » va revenir. Clay Shirky l'avait nommé en 2004 : le logiciel construit pour un petit groupe de personnes spécifiques, résolvant leur problème spécifique, sans ambition de scalabilité universelle. L'ère de l'assemblage avait tué cette idée en rendant les frameworks universels si dominants que personne n'imaginait construire autrement. L'IA générative, paradoxalement, la fait renaître : si un modèle peut générer un outil sur mesure pour un besoin précis, pourquoi acheter un SaaS qui résout 80% de ton problème et te force à contourner les 20% restants ?

Le danger ici est le « vibe coding » — du logiciel situé généré par IA que personne ne comprend et que personne ne peut maintenir. La promesse est le logiciel situé construit avec rigueur — des outils compréhensibles, pour des personnes spécifiques, résolvant des problèmes spécifiques.

3. Les seniors *simplificateurs* vaudront plus qu'avant. Les seniors *gardiens de la dette* vaudront moins. Le marché aura besoin de personnes capables de trois choses que l'IA ne fait pas : comprendre un domaine métier assez profondément pour formuler le bon problème, évaluer si une solution (générée ou non) résout réellement le problème, et maintenir un système sur la durée — c'est-à-dire le comprendre assez pour le faire évoluer sans tout casser. Ce sont des compétences de seniors. Mais de seniors qui simplifient — qui regardent un graphe de dépendances et enlèvent des nœuds, qui remplacent un microservice par une fonction, qui rendent le système compréhensible par la prochaine personne.

Le senior dont le pouvoir vient de l'opacité — le seul à savoir pourquoi le script de déploiement échoue silencieusement le troisième mardi du mois — est menacé de deux côtés. Par l'IA, qui peut éventuellement naviguer dans le marécage de son

code legacy. Et par les juniors, qui avec des outils IA et de la pratique délibérée peuvent atteindre en deux ans un niveau de compréhension structurelle que le rentier n'a jamais cherché à construire, trop occupé à défendre son périmètre. L'IA ne distingue pas l'artisan du péage. Mais le marché, à terme, le fera — parce que le système simplifié par l'artisan coûte moins cher à maintenir que le système rendu opaque par le péage. Le calcul est cruel mais arithmétique.

4. Les juniors qui pratiquent délibérément survivront — les autres seront broyés. Le chemin pour un développeur débutant en 2026 n'est pas « apprendre React et espérer ». C'est : maîtriser un langage au point de pouvoir écrire un left-pad les yeux fermés. Comprendre comment une requête traverse un réseau. Lire du code avant d'en écrire. Construire un projet complet de bout en bout sans framework, puis comprendre pourquoi les frameworks existent. L'IA peut remplacer un assembleur. Elle ne peut pas remplacer quelqu'un qui comprend ce que l'assemblage fait.

5. Le revenu universel deviendra un sujet technique, pas seulement politique. Si 40% des tâches d'externalisation technologique en Afrique sont affectées par l'IA d'ici 2030, et si l'emploi des développeurs juniors continue de chuter, la question du revenu universel de base n'est plus une utopie — c'est un calcul. Les gains de productivité IA vont quelque part. La question est si ce « quelque part » est uniquement les actionnaires, ou si une partie revient aux commoners dont le travail a nourri le modèle. Ce n'est pas une question de générosité. C'est une question de soutenabilité : si vous détruisez la filière de formation (les postes juniors) tout en moissonnant le travail passé (les communs open source), vous n'avez plus de pipeline. La machine mange ses propres graines.

Ce qui ne va probablement *pas* arriver

Le « tout-IA » ne va pas arriver. L'étude METR montre que l'IA ralentit les développeurs expérimentés sur des tâches complexes dans leur propre code. Le code IA contient des vulnérabilités, augmente la dette technique, et crée du « churn » — du code jetable. Les entreprises qui ont massivement adopté le vibe coding vont frapper des murs de maintenance. La correction viendra — douloureusement, comme toujours dans les cycles technologiques — mais elle viendra.

L'open source ne va pas mourir. Les communs survivent aux enclosures — pas intacts, pas indemnes, mais ils survivent. Les communautés du logiciel libre en Afrique francophone, les fondations Apache et Linux, les développeurs qui mettent « no AI training » dans leurs licences — ces gestes ne renverseront pas l'arc, mais ils maintiennent l'espace du possible.

IV. Möbius comme miroir

L'aube

Tu m'as parlé de Möbius. Qu'est-ce que c'est ?

Le crépuscule

Möbius, c'est un langage de programmation à adressage par contenu — chaque morceau de code a une identité dérivée de ce qu'il est, pas de là où il est stocké. Avec un horodatage Bitcoin pour prouver quand il a existé. C'est technique, mais l'idée est simple : un miroir mathématique.

Imagine un système qui te montre trois choses :

Ce que tu as fait. Chaque fonction que tu as écrite a une identité stable et un horodatage. L'historique est incorruptible. Ce n'est pas un CV — c'est une preuve. Tu ne peux pas le gonfler, et personne ne peut te l'enlever.

Ce que tu dois améliorer. Si ton code est dans le graphe, les comparaisons sont possibles. Pas « ton code est meilleur ou pire que celui de X » — mais « voici les patterns que tu utilises, voici les patterns que d'autres utilisent pour le même problème, voici ce qui pourrait être plus simple ». Le miroir ne juge pas. Il reflète.

Ce que nous devons faire pour que tu restes lisible. C'est la partie la plus subversive. Si chaque morceau de code a une identité de contenu, alors le code qui change son nom mais pas sa structure est reconnu comme le même code. Le code qui change sa structure mais pas son nom est reconnu comme différent. La lisibilité — la capacité pour un humain de comprendre ce que fait un système — devient une propriété traçable, pas une aspiration.

L'aube

Et le lien avec l'IA ?

Le crépuscule

L'IA générative produit du code qui fonctionne sans être lisible. Möbius est une réponse possible : un système qui rend la lisibilité traçable. Si le code est adressé par contenu, alors le code généré par IA peut être inspecté de la même manière que le code écrit par un humain — par ce qu'il *est*, pas par qui l'a produit. Le miroir ne se soucie pas de la source. Il se soucie de la structure.

C’est aussi une réponse au problème de l’entraînement. Si chaque morceau de code a une identité et un horodatage, alors la question « ce code a-t-il été utilisé pour entraîner un modèle ? » devient traçable. Pas parfaitement. Pas aujourd’hui. Mais la structure est là.

Et c’est une réponse au problème de l’agentivité. Le principe de Smalltalk dit

> « si un système doit servir l’esprit créatif, il doit être entièrement compréhensible par un seul individu ». Möbius est un système qui rend chaque pièce de code individuellement compréhensible — par son identité de contenu, par sa trace temporelle, par sa place dans le graphe. C’est le contraire exact du vibe coding : non pas « ça marche et je ne sais pas pourquoi », mais « voici exactement ce que c’est, quand ça a existé, et comment ça se relie au reste ».

Et c’est, enfin, une réponse au problème du rentier brownfield. Le miroir ne se soucie pas de qui comprend le code — il rend la structure visible *indépendamment des personnes*. Le rentier perd son pouvoir quand la complexité qu’il gardait opaque devient traçable, mesurable, comparable. Quand le miroir dit « cette fonction a 47 dépendances et pourrait en avoir 3 », le péage n’a plus d’ombre où se cacher. L’artisan, lui, est validé — ses réductions de complexité sont visibles dans le graphe. Möbius rend le simplificateur mesurable et le gardien de dette visible. Ce n’est pas un jugement moral — c’est une propriété structurelle du système.

V. Pratique délibérée : ce que le junior a besoin d’entendre

Le crépuscule

Voici ce que je dirais au junior de 2026, sans condescendance et sans mensonge.

Commence par le domaine, pas par l'outil. Si tu construis un système de facturation, comprends d'abord la facturation. Quels sont les cas limites ? Quelles sont les contraintes légales ? Qu'est-ce qui rend une facture valide dans la juridiction de ton client ? L'outil vient après — et souvent, le bon outil est plus simple que ce que l'industrie te vend. Une base de données relationnelle et un script bien écrit résolvent 80% des problèmes métier. Le reste, c'est de l'ingénierie de la complexité — et l'ingénierie de la complexité a un coût que personne ne t'apprend à calculer.

Apprends un langage au point d'en toucher le fond. Je ne dis pas Scheme parce que je suis partial (je le suis). Je dis : choisis un langage et apprends-le jusqu'à comprendre comment il fonctionne, pas seulement comment l'utiliser. Un langage où tu peux lire l'implémentation. Un langage qui ne te cache pas ses mécanismes. Un langage qui te rend plus intelligent, pas plus productif. La productivité suit la compréhension. Jamais l'inverse.

Construis un système complet sans dépendances externes, une fois. Un serveur web qui écoute sur un port, parse des requêtes HTTP, retourne des réponses. Pas avec Express — à la main. Pas parce que tu vas le faire dans la vraie vie, mais parce que tu ne comprendras jamais ce que fait Express si tu ne sais pas ce qu'il remplace. L'IA peut assembler des appels d'API. Elle ne peut pas te donner la compréhension de ce que ces API font. Cette compréhension est ton avantage concurrentiel — le seul qui résiste à l'automatisation.

Lis du code avant d'en écrire. Le meilleur tutoriel pour apprendre un langage n'est pas un tutoriel — c'est un petit projet bien écrit dans ce langage. Lis-le ligne par ligne. Comprends les décisions. Demande-toi pourquoi l'auteur a fait ce choix plutôt qu'un autre. C'est le même geste qu'un musicien qui transcrit un solo : tu n'apprends pas le solo, tu apprends la pensée derrière le solo.

Résiste à la panique du framework. Quand l'industrie te dit que tu dois apprendre le framework du mois, pose la question : est-ce que ce framework résout un problème que j'ai, ou est-ce qu'il résout le problème de quelqu'un d'autre en créant un nouveau problème pour moi ? Le développeur qui connaît cinq frameworks superficiellement est remplaçable par un modèle de langage. Le développeur qui connaît un domaine en profondeur ne l'est pas.

Et pose-toi cette question chaque jour : est-ce que mon prochain commit rend le système plus lisible ou moins lisible ? Pas plus lisible *pour moi* — plus lisible *pour la prochaine personne*. C'est le test qui distingue l'artisan du rentier. L'artisan simplifie pour que d'autres puissent comprendre. Le rentier complexifie pour que personne d'autre ne puisse comprendre. Les deux peuvent se raconter l'histoire de l'expertise profonde. Mais le code ne ment pas : si ton départ rendrait le système ingouvernable, tu n'as pas construit de l'expertise — tu as construit une dépendance à toi-même. Et l'IA viendra te chercher là aussi, parce que naviguer dans l'opacité, c'est exactement ce que les modèles de langage font de mieux. La seule chose qu'ils ne font pas bien, c'est décider qu'un système est trop complexe et le simplifier. C'est ça, ton travail.

Et si je fais tout ça et que le marché ne me veut quand même pas ?

Alors construis le tien. Le logiciel situé — un outil pour un groupe de personnes spécifiques, résolvant leur problème spécifique — est la chose que l'IA rend plus facile à faire et que le marché ne peut pas te prendre. Pas un SaaS à l'échelle mondiale. Un outil qui résout le problème de la coopérative agricole en face de chez toi, de l'association culturelle de ton quartier, du cabinet médical de ta rue. L'IA peut t'aider à le construire. Mais le domaine — la compréhension de ce dont ces personnes ont réellement besoin — c'est toi qui l'as. C'est le geste qu'aucun modèle ne peut faire à ta place : aller parler à quelqu'un, comprendre son problème, et construire la chose la plus simple qui le résout.

C'est ça, le travail qui vient. Pas l'assemblage de composants sur les conditions de quelqu'un d'autre. La résolution de problèmes réels pour des personnes réelles, avec des outils que tu comprends de bout en bout.

VI. Ce qui vient après l'arc

L'arc que nous avons tracé — de la liberté recadrée à l'enclosure générative — n'est pas une fatalité. C'est une trajectoire. Les trajectoires peuvent être infléchies.

Voici ce qui existe déjà et qui pointe dans une autre direction :

Le logiciel compréhensible. Des langages comme Scheme, des systèmes comme Smalltalk, des architectures comme celle que Möbius explore — des outils où la compréhension n'est pas

sacrifiée à la productivité. Ce sont des niches aujourd'hui. Ils pourraient être des refuges demain.

Le logiciel situé. Des outils construits pour des communautés spécifiques, pas pour un marché mondial. Les communautés du logiciel libre en Afrique francophone — les RAFLL, Ovillage, les JerryMarathons — construisent déjà dans cette direction : des solutions locales, avec des outils libres, pour des problèmes locaux. C'est le contraire exact de la monoculture npm.

La traçabilité comme infrastructure. Le CRA européen exige des SBOM. Möbius propose l'adressage par contenu. Les licences « no AI training » émergent. Ce sont des premiers pas vers un monde où la question « d'où vient ce code, qui l'a écrit, sous quelles conditions ? » a une réponse vérifiable. C'est l'inverse exact de l'invisibilité qui a permis l'arc d'extraction.

La pratique délibérée comme acte politique. Mais attention : la pratique délibérée *vers la simplification* est l'acte politique. La pratique délibérée *de la complexité* est la rente. Le premier geste rend le système compréhensible par d'autres — c'est un bien public. Le second rend le système dépendant de soi — c'est une capture privée déguisée en expertise. Le développeur qui refuse le vibe coding comme mode par défaut et qui simplifie ce qu'il touche maintient une capacité collective : la capacité de la société à comprendre les systèmes dont elle dépend. Le développeur qui empile les abstractions pour sécuriser son poste fait le contraire — il produit de la dette technique qui, un jour, sera soit absorbée par un modèle de langage, soit abandonnée. Ni l'un ni l'autre ne le sauvera. Drucker avait raison : les plans ne sont que de bonnes intentions tant qu'ils ne dégénèrent pas en travail concret. Le travail concret ici, c'est chaque commit, chaque revue de code,

chaque décision d'architecture. Le test tient en une question : est-ce que quelqu'un d'autre que moi peut comprendre ce que je viens de faire ?

VII. La dernière conversation

L'aube

Si tu devais résumer tout l'arc en une phrase, pour quelqu'un qui n'a jamais écrit une ligne de code ?

Le crépuscule

Les gens qui ont construit les fondations numériques du monde l'ont fait en partageant leur travail. D'autres ont clôturé ce travail et l'ont vendu. Puis ils ont utilisé le travail lui-même pour fabriquer des machines qui remplacent les personnes qui l'avaient fait. Maintenant, la question est : est-ce qu'on accepte ça comme le prix du progrès, ou est-ce qu'on construit autrement ?

L'aube

Et la réponse ?

Le crépuscule

La réponse n'est pas une technologie. C'est une pratique. Construire des choses que tu comprends, pour des personnes que tu connais, avec des outils que tu peux inspecter. Et laisser une trace — pas une trace dans un modèle statistique, mais une trace lisible, versionnable, adressée par contenu, ancrée dans le temps.

Pas que tout le monde pense la même chose. Mais que la pensée de chacun laisse une trace.

L'aube

C'est ambitieux.

Le crépuscule

Non. C'est simple. C'est pour ça que c'est difficile.

Ce texte est le prologue et l'épilogue d'une trilogie : « Avant les plateformes » (ère 1), « L'ère de l'assemblage » (ère 2), « L'ère générative » (ère 3). Les quatre documents ensemble — le prélude, les trois ères — composent un arc complet, du laboratoire du MIT aux prompts de 2026. Ils peuvent être lus dans l'ordre ou dans le désordre; chacun se tient seul, mais les quatre ensemble racontent une histoire que les parties séparées ne racontent pas.

← L'enclosure du code — Série complète

möbius - 3.01 - the invariant - the hold

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/01-the-hold/

möbius - 3.02 - the invariant - the garage lab hacker

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/02-the-hacker/

möbius - 3.03 - the invariant - the polyglot programmer

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/03-the-polyglot/

möbius - 3.04 - the invariant - the burned-out senior dev

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/04-the-senior/

möbius - 3.05 - the invariant - the kid in tizi ouzou

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/05-the-kid/

möbius - 3.06 - the invariant - the pl theorist

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/06-the-theorist/

möbius - 3.07 - the invariant - the coordinator

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/07-the-coordinator/

möbius - 3.08 - the invariant - the civilizational thinker

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/08-the-thinker/

möbius - 3.09 - the invariant - the air cycle

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/09-the-air-cycle/

möbius - 3.10 - the invariant - the check

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/10-the-check/

möbius - 3.11 - the invariant - departure

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/11-departure/

möbius - 3.12 - the invariant - the privateer

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/12-the-privateer/

möbius - 3.13 - the invariant - the citizen

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/13-the-citizen/

möbius - 3.14 - the invariant - the educator

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/14-the-educator/

möbius - 3.15 - the invariant - the connector

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/15-the-connector/

möbius - 3.16 - the invariant - the symbiotic ai

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/16-the-symbiotic/

möbius - 3.17 - the invariant - the anchor backlog

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/17-the-anchor-backlog/

möbius - 3.18 - the invariant - the refactor cascade

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/18-the-refactor/

möbius - 3.19 - the invariant - the memory wipe

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/19-the-memory-wipe/

möbius - 3.20 - the invariant - the artist

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/20-the-artist/

möbius - 3.21 - the invariant - the silence

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/21-the-silence/

möbius - 3.22 - the invariant - the person who was stolen from

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/22-the-stolen/

möbius - 3.23 - the invariant - the independent researcher

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/23-the-researcher/

möbius - 3.24 - the invariant - the community steward

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/24-the-steward/

möbius - 3.25 - the invariant - the academic researcher

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/25-the-academic/

möbius - 3.26 - the invariant - the representative

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/26-the-representative/

möbius - 3.27 - the invariant - the manifesto

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/27-the-manifesto/

möbius - 3.28 - the invariant - the invariant

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/28-the-invariant/

möbius - 3.29 - the invariant - the language born in transit

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/29-the-language/

möbius - 3.30 - the invariant - the first commit

2026 / 04 / 26 · https://hyper.dev/mobius/the-invariant/30-the-first-commit/

möbius - 5 - a content-addressed Scheme that speaks your language

2026 / 05 / 06 · https://hyper.dev/2026/mobius-bb-scm/

Today I'm tagging **bb.scm v0.1.0** — the first implementation release of Möbius. For the project vision, see Möbius Product Brief. bb.scm is a Chez-Scheme implementation of a content-addressed programming language where code identity is structural, not nominal. Same algorithm in Tamazight, Arabic, French, English → identical SHA-256 hash, after de Bruijn normalization. Names are mappings. The hash underneath is the same.

What "v0.1.0" means

The full design lives in `manual.md`. v0.1.0 is the part of that design that runs.

What is done

The language (RºRM, Part II). A tree-walking interpreter for the 39 foundations and the one mechanism — `gamma`, the catamorphic pattern-matching combiner. The reader, the de Bruijn normalizer, the pattern matcher, the environment frames, capsules, boxes, lambda, guard/continuation foundations, and a base library written in Mobius itself. SHA-256 in pure Scheme so identity is reproducible without external dependencies.

The store (Part III, partial). Append-only content store on the filesystem: `combiners/<sha256>/tree.scm` for the immutable normalized AST, `mappings/<hash>/map.scm` for per-language

presentation, plus lineage, worklog, and review marks. `bb add` parses, normalizes, hashes, and stages. `bb commit` promotes WIP to committed. `bb refactor` propagates a hash substitution through a dependency tree. `bb diff` does a structural compare with ANSI colors. `bb validate` re-hashes the whole store.

The toolchain (Part IV). Most of `bb`'s command surface from §29 of the manual:

- *Authoring:* `add`, `edit`, `commit`, `refactor`
- *Verification:* `check`, `review`, `validate`, `diff`
- *Navigation:* `show`, `search`, `tree`, `caller`, `resolve`, `log`, `status`
- *Evaluation:* `eval`, `run`, `repl`, `print`
- *Distribution:* `remote add/list/remove/push/pull/sync`, `remote publish/stop`
- *Housekeeping:* `store init`, `store info`, `worklog`, `mapping list/set/delete`

A runtime around the language. A small `transparent` runtime (io_uring HTTP server/client) ships alongside, so combiners can be served without leaving the Mobius surface — see `bb serve <app-ref> <handler-ref>`.

Reference syntax. A flexible `<ref>` grammar — `name`, `name@nid`, `name@nid@lang`, `name@nid@lang@lid` — with hash prefixes (minimum unique) accepted anywhere a hash is expected. Documented in the README and the project CLAUDE.md.

What is not done

Read the bottom half of the manual as the to-do list. Concretely:

- `bb prove` **and** `bb verify` **(§29.8).** Zero-Knowledge Proofs for sealed combiners — the *Privateer* persona. The seal

works (hashes can be locally committed and OTS-anchored without disclosure); proving capability without disclosure does not.
- **Oblivious execution (§31).** Orthogonal to Möbius and explicitly not in scope for the toolchain at this stage. The chain *Coordinator* → *Privateer* → *Citizen* breaks at the runtime-privacy step until ORAM/TEE/MPC integration is designed.
- **Atlas Stoa (§32).** The read-only aggregation layer above stores: structural near-duplicate detection (`bb search --near`), behavioral indexing by check suite, language-axis search across federated stores. Designed; not built.
- **The 15 open questions of §33.** Notably: a specified error model, effects as named patterns over continuations, `eval` as `integer->combiner` hash lookup, top-level mutual recursion, capsule type-ID derivation, parameterized hash/anchor/proof-system, and predicate inference. These are real and not hand-waved.
- **The ellipsis (Annex A).** Listed as a possible future addition, not part of v0.1.0.

The honest summary: the language and the store are usable today; the cryptographic-trust layer (ZKP + oblivious execution) and the cross-store discovery layer (Atlas Stoa) are designed but not implemented; the formal corners (effects, errors, eval) are open questions, not bugs.

The examples

Three example projects ship under `examples/`. They are not toys — they exist to stress the language at the boundary where the multilingual story has to survive contact with HTTP, mutable state, and a real browser.

`examples/counter` — the clicker

A server-rendered click-counter that survives page refreshes. Demonstrates the two-combiner server interface — `bb serve <app-ref> <handler-ref>` — where `app-ref` is called once at startup to build shared state and `handler-ref` runs on every request. State is a single `box` holding a Chez vector; mutation is `box!`. The interesting part is `xeno`: the escape hatch from Mobius into Chez. It's there, it's necessary at the I/O boundary, and the design pressure is to keep it small.

`examples/todomvc` — the canonical exercise

TodoMVC, but as Mobius combiners served by the same `bb serve` runtime. Eight combiners — `todos`, `todos-state`, `todos-toggle`, `todos-delete`, `todos-remove-completed`, `todos-count-active`, `todos-render-list`, `todos-render-item` — composed by hash, deployed as one process. The HTTP request/response shape is an alist of strings, the state is `(cons next-id todos-list)`, and the handler dispatches on `method` + `action` query param. It exists because TodoMVC is the lingua franca of "is your stack real?"

`examples/invariant` — the trilingual blog

A full static site generator written in Mobius — *trilingual* en/fr/es — generating the Invariant blog. Thirty stories per language. The user-facing combiners share a hash across language mappings: `generate` / `engendrer` / `engendrar`, `render-post` / `afficher-recit` / `mostrar-relato`, and so on. The `xeno` surface is two files, both in `src/io/`. Everything else — list traversal, string composition, HTML construction, language dispatch — is pure Mobius.

This is the example that matters most to me. It is the meta-story of the project: the personas of the manual are recurring

characters in the fiction, the store is the fictional infrastructure, and the blog is generated by the system the blog is about. Two real bugs in `bb` were discovered and fixed during its construction (mapping resolution order; qualified name self-reference for ambiguous combiners). Dogfooding earned its keep.

How to try it

The bb.scm repository is available on GitHub: github.com/amirouche/bb.scm

```
git clone https://github.com/amirouche/bb.scm
cd bb.scm/north
./bb repl
```

Or evaluate the abacus from the README — six clauses of catamorphic arithmetic — in three languages, three identical hashes:

```
(define abacus
  (gamma
    (("num" ,n) n)
    (("add" ,(a) ,(b)) (+ a b))
    (("sub" ,(a) ,(b)) (- a b))
    (("mul" ,(a) ,(b)) (* a b))
    (("neg" ,(a)) (- 0 a))
    (("do"  ,(r))    r)))
```

The recursion is in the commas. The identity is in the hash.

A taste of the check language

A check is a combiner that takes the combiner under test as its argument. `bb check` applies the check to the candidate; `assume` decides pass/fail.

Hard-coded check — concrete inputs, concrete expected outputs. Useful for base cases and regression tests:

```
(define ~check-abacus-base-cases
  (lambda (abacus)
    (begin
      (assume (= 7  (abacus (cons "num" (cons 7 #nil))))
              "literal 7 evaluates to 7")
      (assume (= 5  (abacus (cons "add"
                                  (cons (cons "num" (cons 2 #nil))
                                    (cons (cons "num" (cons 3 #nil)) #nil)))))
              "2 + 3 = 5")
      (assume (= -4 (abacus (cons "neg"
                                  (cons (cons "num" (cons 4 #nil)) #nil))))
              "neg of 4 is -4"))))
```

Three concrete trees, three concrete answers. Run `bb check abacus` and it prints `PASS` / `FAIL` per assumption.

Symbolic check (Z3) — return a lambda instead of calling `assume` directly. Its parameters become Z3 variables, its body becomes an SMT assertion, and the combiner under test is inlined as a `define-fun`. The property is verified universally over the parameter domain, not on a finite witness set:

```
(define ~check-double-is-times-two
  (lambda (double)
    (lambda (x)
      (assume (= (double x) (* 2 x))
              "double(x) = 2 * x for every integer x"))))
```

When `z3` is on `PATH`, `bb check` reports `Z3-PASS` (the SMT solver found no counterexample) or `Z3-FAIL` with the offending value of `x`. When `z3` is missing, the same combiner returns `Z3-SKIP` and the rest of the suite still runs — symbolic checks degrade gracefully into “not verified here.”

The two styles compose: a check suite typically pins specific corner cases by hard-coding and proves the general invariant symbolically. The check suite *is* the specification — content-addressed, forkable, translatable into Tamazight or French, and (eventually, when `bb prove` lands) the statement a Zero-Knowledge Proof would attest to without disclosing the implementation.

tl;dr

First release of bb.scm — a CLI for Möbius with Chez Scheme

Content-addressed "Scheme": write code, get a hash, the same hash across French, English, Tamazight, Arabic, and Mandarin.

bb.scm on GitHub:

```
git clone https://github.com/amirouche/bb.scm
cd bb.scm && make
./bb --help
```

Feedback welcome.

möbius - 6 - Product Brief

2026 / 05 / 08 · https://hyper.dev/2026/mobius-product-brief/

Executive Summary

Mobius is a proof by construction. The thesis: the Hobbesian equilibrium — where defection is rational and cooperation is naive — is not a fact of human nature. It is a structural consequence of fog. Remove the fog, and cooperation becomes the natural equilibrium. Not because people are good. Because the structure makes defection unprofitable.

The fog has concrete mechanisms: opacity of provenance, language as a power asymmetry, access barriers that filter by privilege rather than ability, and attribution systems controlled by the institutions that benefit from asymmetry. Every existing software and knowledge infrastructure was built within this fog, and reproduces it.

Mobius replaces the fog with a transparent substrate: content-addressed code where priority is established by cryptographic proof rather than institutional permission, where the same algorithm in Tamazight and English produces an identical hash, where derivation is structural and cannot be quietly erased, and where the barrier to participation is 43 primitives and one mechanism — not a 400-page manual, not git, not an institutional affiliation.

The primary beneficiary is the Tizi Ouzou kid: someone with knowledge and creativity who cannot currently participate in the global knowledge commons because every path through it was built by and for others. Mobius is built so that when they

pick it up, it works — and their contribution is theirs, verifiably, permanently, in their language.

Core Vision

Problem Statement

The dominant cosmology of software and knowledge infrastructure assumes defection as the natural equilibrium. This assumption is self-fulfilling: systems built on it create the fog that makes defection rational. The fog operates through opacity of provenance (who wrote what, when — unverifiable without trusting an institution), language asymmetry (knowledge production requires English fluency and Western development norms), access barriers as gatekeeping (git, build chains, package ecosystems as filters of privilege), and attribution systems controlled by those who benefit from asymmetry. The result: the Tizi Ouzou kid — knowledgeable, creative, willing — cannot participate in the global knowledge commons on equal footing.

Problem Impact

When the fog persists, knowledge production remains concentrated among those who already hold institutional access. Ideas are stolen without recourse. Priority is assigned by whoever has legal infrastructure, not whoever had the insight. Entire languages and cultures are effectively excluded from participation. The cost is not merely individual injustice — it is the civilizational cost of knowledge that never gets created, problems that stay unsolved, contributions that are made and then erased.

Why Existing Solutions Fall Short

GitHub, npm, and open source require English fluency, git expertise, and Western development norms. Academic publishing requires institutional affiliation and peer review gatekeeping. Patents require legal infrastructure, money, and jurisdictional access. Blockchain-based attribution systems exist but are speculative markets, not knowledge infrastructure. All of these operate within the Hobbesian framework: they are gates, and the gates were built by and for those who already hold access. None of them make the fog structurally impossible — they route around it for the privileged, while leaving it intact for everyone else.

Proposed Solution

Mobius is a programming language, a content-addressed store, and a toolchain that together constitute a transparent substrate for knowledge. Code is stored as language-independent de Bruijn trees — the same algorithm in any human language produces the identical hash. Priority is established by Bitcoin-anchored OpenTimestamps proofs, requiring no institution and no permission. Derivation is structural: the lineage of every function is in the content itself, not in a registry someone controls. Participation requires learning 43 primitives and one mechanism. The `bb` toolchain replaces git for knowledge sharing: send a hash, map it to a name in your language, done.

The Möbius strip has no inside or outside — it is the same surface throughout, with no asymmetric position to exploit. That is the geometric argument the project makes concrete.

Key Differentiators

Priority by construction, not permission. A Bitcoin-anchored hash is not a claim — it is proof. No institution can grant it; none can revoke it.

Language as a first-class citizen. The same algorithm in Tamazight, Arabic, French, and English produces the same hash. Multilingualism is not an afterthought or a translation layer — it is the architecture.

Derivation is structural. The lineage of every function is in `tree.scm`, not in a mutable registry. Adversarial translation cannot erase it.

Minimum viable barrier. 43 primitives, one mechanism (`gamma`). The entire language fits in a person's head. No gatekeeping through complexity.

A proof, not a proposal. Mobius does not argue that a better cosmology is possible. It demonstrates it. The infrastructure is the argument.

A mathematical mirror. Mobius is only and no more than a mathematical mirror of what we live and believe. Code is data. Data is code. A constant can be a number, a story, an ancient oral tradition. A function can be executable code, code poetry, a knowledge graph relation. It does not need to run. It needs to tell a story.

Target Users

Primary Users

The Garage Lab Hacker

An engineer's version of the Independent Researcher. Works alone or in small clusters — a bedroom, a shared workshop, a library in Tizi Ouzou with an Orange Pi. Solves small problems that cause large pain: a bug in directory listing, a missing function, a path no one has documented yet. Their contributions accumulate — small and not-so-small, little by little. Their defining strengths are **vision of a better future and endurance**. They are not sprint heroes. They are long-distance builders who see where the road needs to go and are willing to walk it for years.

Current situation: They publish to GitHub under their own name, hope for the best. Priority is established by whoever has the most followers or the right institutional affiliation. Their work gets borrowed without credit and they have no recourse. The fix disappears into the fog.

What Mobius gives them: A hash is theirs. Bitcoin-anchored at the moment they publish — no institution required, no permission needed. The content itself is the proof. Their contribution doesn't need to be discovered immediately. It accumulates. It is theirs while it waits.

Discovery path: A library or cyber kawa with a pre-configured Orange Pi or Raspberry Pi — Linux, `bb`, and git already installed. A local or public forge (Codeberg, SourceHut, GitHub, self-hosted). Existing Mobius code to learn from. `bb store init`, write a first function, run checks, see them pass. The feedback loop is immediate. No configuration, no dependency resolution, no build step.

Relationship with existing infrastructure: They still push to GitHub — because that's where the community is, because it scratches an itch, because it documents a little-known path.

Mobius layers beneath: identity, priority, derivation — the things GitHub never had and never claimed to have. The two do not conflict. They compound.

Success: Not a single moment. The accumulation itself. Watching the infrastructure grow around their contributions. Knowing the work is theirs, verifiably, permanently, in whatever language they named it.

The Burned-Out Senior Dev

Ten or fifteen years in. Deep expertise. Knows better than anyone what needs to be fixed and how. Their orientation is not frustration seeking relief — it is **stewardship**. They work to leave the place better than when they arrived. If there is a chance that in 5 to 10 years they can take advantage of their contribution to Mobius in their daily work, that will be great. If not, they will rest assured they did the right thing. The act of contribution is its own reward.

Current situation: They have learned to work around the machinery — merge conflicts, dependency hell, version drift, the political overhead of large codebases where trust erodes and every refactor requires negotiation. They manage it. Managing it is itself a cost.

What Mobius gives them: Immutable content. No merge conflicts — content is addressed by hash, not file. No "everyone must upgrade" — old hashes keep working until you're ready. `bb refactor` updates one path in the DAG; everything else stays on the old version. The working session is `bb edit freedom@lang` — no branching strategy, no merge workflow. Old code isn't debt unless it's actually broken.

Why they come: Not feature comparison. They recognize a project worthy of their investment. The proof-by-construction framing of Mobius aligns structurally with a stewardship ethic: someone building infrastructure so the place is better for whoever comes after. They understand that story from the inside.

Relationship with existing infrastructure: They still commit to GitHub, still work in their existing stack. Mobius is where the things they fix get their permanent proof of existence. The contribution layers, it doesn't replace.

Success: First `bb refactor` where content-addressing resolves a coordination problem they'd accepted as permanent. And the longer arc: knowing their contribution is in the foundation, regardless of whether they're still there to see it used.

The Tizi Ouzou Kid *(aspirational primary beneficiary — the proof)*

Knowledgeable, creative, capable of contributing to the global knowledge commons. Currently locked out — not because they lack ability, but because every path through the existing ecosystem was built by and for someone else. English fluency required, git knowledge required, institutional affiliation required, Western payment infrastructure required.

They are not an edge case. They are the **central case** — the test of whether the cosmological argument is true. If they can participate on equal footing, in Tamazight, from a library with an Orange Pi, without institutional permission, then Mobius has demonstrated that the Hobbesian world was a consequence of fog, not of nature.

They are not a passive recipient of infrastructure others built. They are a node in a knowledge network that was always already theirs — they simply didn't have a way to be legible in it. When they encode a constant, write a function, share a story in round syntax, they are not being granted access. They are claiming what was always true: that their knowledge belongs to them and to the commons simultaneously.

The proof moment: When the Tizi Ouzou kid publishes a function in Tamazight, receives a Bitcoin-anchored proof, and is referenced by someone in another country who maps it to their own language — the argument is no longer theoretical.

Secondary Users

The Connector — makes the network more valuable for everyone else. Finds relationships between contributions across languages, contexts, and communities. Benefits from semantic search across embedding axes, the ability to trace derivation across mappings, findability of functions by what they do rather than what they're called. Their work amplifies every contribution in the store.

The Symbiotic AI — uses the Mobius commons as external memory. Content-addressed, cryptographically verified, derivation-traceable knowledge is exactly what a trustworthy AI substrate looks like: the hash is the trust anchor, derivation cannot be fabricated, the multilingual structure means the AI works across languages without the asymmetry that monolingual corpora produce. `bb skill` is the interface that gives an AI system enough context to work with the store directly.

The Boundary

Mobius is not for those whose advantage depends on the fog persisting. Not a demographic exclusion — a structural one. The transparent substrate removes what asymmetry was living off of.

User Journey

Discovery: Pre-configured hardware in libraries and cyber kawas. For the senior dev: a post or talk that names the friction they've been tolerating and recognizes the stewardship ethic underneath it.

Onboarding: `bb store init`. Write a first function — or a first constant. A number, a name, a story. Run checks if it's executable. The hash is computed. The content is theirs.

Core usage: `bb edit` for the working loop. `bb commit` as the deliberate act. `bb publish` to establish priority. `bb search` and `bb search --near` to find related knowledge. `bb log` to see the full timeline of what has been built.

Success: Not a single moment. The accumulation. The infrastructure growing. The strongly connected components forming over time — not by design, but by social gravity and shared problems. Near-zero cost colocation means the accumulation is global from the first contribution, but clustering happens later, organically. It is constructed, not launched.

Scope: Code is data. Data is code. A constant can be a number, a story, an ancient oral tradition. A function can be executable code, code poetry, a knowledge graph relation. It does not

need to run. It needs to tell a story. Mobius is a mathematical mirror of what we live and believe — and the mirror holds everything that can be expressed.

Success Metrics

Mobius operates on a civilizational timescale. Success is not optimized for — it is observed. The signals below are what we expect to see if the infrastructure is working. They are not targets. Optimizing for them would distort the mirror.

The Substrate Is Already Enough

Before any signal, before any accumulation: the substrate is useful today, at the scale of one person, without requiring a network, without requiring the cosmological proof to be demonstrated. A Garage Lab Hacker who fixes a bug and publishes a hash has succeeded. A Senior Dev who contributes one well-considered function to the commons has succeeded. A parent who leaves their children the beginning of a map has succeeded.

Massification may follow. The civilizational signals may emerge. But accumulation is not the condition of usefulness. The substrate works now. That is already enough.

Substrate Health vs. Output Signals

Two distinct measurement categories. Two different interventions.

Substrate health is an engineering responsibility — UI, UX, code, data. Is `bb edit` fast enough? Is the forge accessible from a library in Tizi Ouzou? Is the onboarding friction low enough

that the first hash is achievable in an afternoon? These are things to maintain and improve. They are measurable without distorting the output signals.

Output signals are what the mirror shows. They are observed, never optimized for. The moment you target an output signal, you corrupt it: the number appears without the thing. Watch, don't aim.

Silence signals also carry information. If the store is empty after two years, that is not neutral — it means the substrate needs attention. But the distinction between "broken substrate" and "civilizational patience" is answerable: if the substrate serves the builder, it is probably healthy. If it doesn't serve even the person who built it, it is broken.

What Success Looks Like at 2 Years

If the infrastructure is working, we expect to observe:

- Approximately 100 small tools in the Mobius store — written from scratch or ported. Both count. Both exercise the mirror. The ratio between them is the first visible signal: whether people are arriving to innovate, to preserve, or both. It will not be designed. It will be shown.
- Individual git repositories per tool, authored independently.
- At least one community repository aggregating contributions from multiple authors.
- At least one Cosmic Ledger instance serving as a read-only, queryable, composable commons — not a museum, but a living infrastructure. Like Software Heritage, but for living projects.

- At least one function published in a non-English language mapping.
- At least one Bitcoin-anchored proof per active contributor.

What Success Looks Like at 10 Years

An independent researcher — unaffiliated, without institutional backing — publishes:

- A paper: the formal claim, derivation traceable to content-addressed hashes.
- A poster: the accessible summary.
- A talk: the public presentation.
- A bootable container: the complete artifact. Anyone can run it to verify her claims, reproduce her results, and investigate her reasoning.

No institution granted her this. The hash is her institution. The bootable container is the proof that her work stands on its own. This scenario cannot be manufactured. She either arrives — because the infrastructure made it possible — or she doesn't.

What Success Is Not

Infrastructure is measured by what it makes possible — not by how often people look at it.

Mobius does not have KPIs. It has signals. The signals appear naturally if the substrate is right. They cannot be optimized for without becoming false.

Watch the mirror. Don't point it at itself.

The Design Constraint That Holds Everything

43 primitives and one mechanism is not only pedagogical economy. It is a decision about how much of a person's life a tool is allowed to consume. The substrate must not be heavier than the life that carries it.

The worst corruption of Mobius would be twofold: that it disconnects people instead of connecting them, and that the technical work consumes the builder to the point of leaving aside family and friends. Mobius is built to leave the place better — but the place includes the people at home.

The self-sustaining questions — *Is a sufficiently smart compiler possible? Is partial evaluation the most elegant path? Can we collapse the tower of interpretation? What connects us?* — are the open frontier that keeps the work alive. They generate interesting problems faster than they resolve them. That is the architecture of a decade of sustained curiosity.

Mobius is not a manual. It is the beginning of a map.

Near Term Success Criteria

The v0.1.0 is successful when Intent uses it in their own work without friction — when `bb edit`, `bb commit`, `bb refactor`, and `bb run` form a working loop that serves daily use in practice, not just in design. No external validation required. The substrate serves the person who built it. That is the first and most demanding test.

Future Vision

The initial release establishes the authoring loop. What follows, in rough order of dependency:

- **Publishing layer:** `bb publish` with OTS timestamps, `bb sync` — priority becomes cryptographic
- **Discovery layer:** `bb search`, `bb log`, `bb status` — the store becomes navigable
- **Trust layer:** `bb review` — explicit, cumulative, auditable trust
- **Semantic layer:** embeddings, `bb store index`, `bb search --near` — functions findable by what they do
- **Cultural layer:** multiple active language mappings, community Cosmic Ledger — the commons legible across languages
- **Proof layer:** the independent researcher scenario — the cosmological argument demonstrated

Each layer rests on the one before it. The bb.scm v0.1.0 (GitHub) is the first step onto the map, following the vision described in Toward An Ecology Of Memory and Computation.

möbius - 0 - Toward An Ecology Of Memory and Computation

2026 / 05 / 15 · https://hyper.dev/2026/mobius-toward-an-ecology-of-memory-and-computation/

We no longer suffer from a lack of information. We suffer from the cost of maintaining meaning across time.

Modern civilization produces an unprecedented quantity of code, text, media, models, datasets, interpretations, and synthetic artifacts. Yet the dominant infrastructures underneath this abundance remain strangely fragile:

- provenance disappears,
- transformations become opaque,
- maintenance becomes invisible,
- knowledge fragments,
- institutions centralize memory,
- language becomes a gatekeeping layer,
- and increasingly, humans consume outputs they cannot realistically audit.

The crisis is not only technical. It is cognitive, cultural, and civilizational.

Möbius emerges from the intuition that many of these pathologies share a common root: our infrastructures optimize for production more than lineage.

We became extraordinarily good at generating artifacts. We remained comparatively poor at preserving:

- where they came from,
- how they evolved,

- what they refine,
- what they depend on,
- who maintains them,
- what survives them,
- and which transformations can still be trusted.

The result is a civilization increasingly surrounded by computational dark matter: knowledge exists, but becomes difficult to traverse, verify, refine, or transmit.

Möbius does not begin with the dream of a perfect society.

It begins with a simpler hypothesis:

> perhaps some forms of coordination become less costly if the substrate itself changes.

Not through centralization. Not through universal governance. Not through a final protocol capable of resolving politics, truth, or conflict.

But through modifications to the properties of the medium itself.

At its core, Möbius treats knowledge less as static documents and more as living computational lineage:

- append-only,
- forkable,
- executable where appropriate,
- refinable,
- and distributable.

In this model, the hash is not the point. The lineage is.

An isolated artifact has little value on its own. What matters is:

- what it refines,
- what depends on it,
- who maintains it,
- which communities trust it,
- which predicates it satisfies,
- which paths make it reachable.

The system does not attempt to eliminate plurality. It attempts to make plurality composable.

Forkability is not merely a software property. It is the civilizational ability to continue when coordination fails.

This distinction matters.

Many technological visions become implicitly messianic. They begin as infrastructures and slowly evolve into theories of civilization itself:

- the correct governance,
- the correct market,
- the correct protocol,
- the correct intelligence,
- the correct social order.

Möbius attempts something narrower and perhaps stranger.

It does **not** assume:

- consensus,
- alignment,
- universal transparency,
- or even complete intelligibility.

It assumes instead that the future will likely remain:

- heterogeneous,
- partially opaque,
- forked,
- multi-agent,
- and cognitively uneven.

Some knowledge will remain public. Some will remain sealed. Some actors will cooperate. Others will speculate privately. Some registries will converge. Others will diverge permanently.

The system does not interpret this as failure. It treats it as ecology.

Not ecology in the sense of technological purity or innocence, but in the older sense: a study of relations between organisms and their environment.

Applied cognitively, the question becomes:

- how ideas survive,
- circulate,
- compete,
- decay,
- compose,
- mutate,
- and consume resources across time.

This framing matters because modern computational systems are not immaterial.

Large-scale AI systems consume:

- electricity,
- hardware,
- water,

- mineral extraction,
- supply chains,
- centralized compute infrastructure.

There is no pure digital world floating outside material reality.

Möbius does not attempt to erase this contradiction.

Instead, it asks a different question:

> which informational ecosystems generate less destructive entropy over long timescales?

Because informational disorder also has a material cost:

- duplicated labor,
- unverifiable synthesis,
- abandoned ecosystems,
- institutional memory collapse,
- non-maintainable stacks,
- opaque transformations,
- fragmented knowledge,
- and slop without lineage.

Civilizations pay for informational architectures whether they notice it or not.

The Skeptic

A skeptical reading of Möbius remains necessary.

Not because skepticism invalidates the project, but because infrastructures capable of reshaping informational environments always risk becoming larger than their creators initially intend.

The skeptic asks:

- does lineage eventually become bureaucracy?
- does verifiability privilege already-equipped actors?
- do registries become new centers of soft power?
- does executable knowledge marginalize tacit knowledge?
- does maintenance visibility become a new form of social pressure?
- does cognitive compression create new forms of opacity elsewhere?
- does distributed infrastructure merely relocate centralization into tooling and compute?

The skeptic also reminds us that civilizations do not merely run on truth or verification.

They also run on:

- ambiguity,
- forgetting,
- myth,
- narrative,
- approximation,
- secrecy,
- and selective illegibility.

Some knowledge survives precisely because it resists formalization.

Others survive because communities protect them from optimization.

Not all knowledge survives through formalization.

Civilizations also depend on tacit, embodied, local, emotional, ritualized, and socially transmitted forms of intelligence:

- craft traditions,
- oral memory,
- aesthetic judgment,
- care work,
- improvisation,
- political intuition,
- and cultural practices whose meaning cannot be fully separated from the communities carrying them.

A civilization that attempts to encode everything risks sterilizing the very conditions that allow living cultures to adapt.

Möbius therefore should not be interpreted as an attempt to replace tacit knowledge with executable systems, but as an attempt to make certain forms of lineage, refinement, and maintenance more traversable where formalization is beneficial.

Not all opacity is pathology.

Some forms of opacity protect:

- dignity,
- autonomy,
- local meaning,
- cultural continuity,
- strategic sovereignty,
- and the right not to be fully translated into external systems.

Édouard Glissant's notion of *droit à l'opacité* — the right to opacity — poses one of the deepest philosophical tensions for lineage-oriented civilizations.

A society organized around traversable derivation and visible refinement must confront an uncomfortable possibility: complete legibility may itself become a form of domination.

Historically, systems of power often expanded through:

- classification,
- mapping,
- forced translation,
- administrative visibility,
- and the reduction of plural realities into externally navigable schemas.

Möbius does not resolve this contradiction.

It merely attempts to create environments where:

- visibility,
- secrecy,
- refinement,
- opacity,
- and forkability

remain negotiable rather than structurally imposed.

From this perspective, Möbius should not be interpreted as an attempt to eliminate ambiguity from civilization.

It is better understood as an attempt to reduce unnecessary informational friction while preserving the possibility of plurality, opacity, and forks.

The distinction matters.

Without it, every infrastructure slowly drifts toward imperiality.

Möbius therefore sits in tension between two very different speculative traditions.

On one side lies the distributed abundance imagined by Iain Banks in The Culture series: civilizations shaped by advanced intelligences, vast computational coordination, and post-scarcity infrastructures.

On the other side lies the anthropological skepticism of Ursula K. Le Guin: a persistent attention to power, exclusion, tacit knowledge, cultural opacity, and the human cost of systems that become too coherent.

Möbius does not fully belong to either tradition.

It explores the possibility that highly computational civilizations might remain:

- pluralistic,
- forkable,
- partially opaque,
- and resistant to total closure.

Cheap Power

Every civilization is shaped not only by its values, but by the forms of power it makes inexpensive.

Feudal societies made hereditary power cheap. Industrial societies made mechanical production cheap. Networked societies made distribution cheap.

The digital era dramatically reduced the cost of copying information while leaving verification, provenance, and long-term maintenance comparatively expensive.

This imbalance produced strange incentives:

- visibility became easier than stewardship,
- synthesis became easier than understanding,
- publication became easier than maintenance,
- generation became easier than verification.

Large-scale AI systems accelerate this imbalance even further.

When content generation becomes nearly free, civilization risks drowning in artifacts whose lineage, assumptions, dependencies, and transformations become increasingly difficult to audit.

In such environments, opacity itself becomes economically productive.

The Hobbesian fog thickens.

Actors capable of navigating ambiguity gain disproportionate leverage:

- platforms,
- institutions,
- brokers,
- credential systems,
- opaque infrastructures,
- and entities able to centralize interpretation.

Möbius explores the possibility of shifting this economic terrain.

Not by removing power.

But by changing which forms of power become cheap.

If executable lineage, distributed verification, composable refinement, and visible maintenance become less costly, then some existing asymmetries weaken:

- coordination monopolies,
- archival monopolies,
- provenance monopolies,
- and certain forms of institutional opacity.

At the same time, new asymmetries emerge:

- compute access,
- tooling sophistication,
- optimization control,
- cognitive navigation capacity,
- registry influence,
- and attention concentration.

The fog does not disappear. It changes shape, and parts of it become visible.

This is why Möbius should not be understood as a blueprint for a frictionless civilization.

A frictionless civilization may be impossible.

The more meaningful question is:

> which frictions remain, and who benefits from them?

This is why the project gravitates toward concepts such as:

- local-first infrastructures,
- append-only refinement,
- multilingual registries,

- executable artifacts,
- distributed maintenance,
- and heterogeneous intelligences.

Not because these properties create utopia.

But because they may reduce certain forms of civilizational friction:

- duplicated labor,
- unverifiable synthesis,
- maintenance collapse,
- institutional memory loss,
- linguistic exclusion,
- hidden dependencies,
- and coordination costs amplified by opaque systems.

Möbius is therefore less a blueprint for a future society than a proposal for a different informational climate.

A different ecology of memory.

In such an ecology, maintenance becomes more visible than novelty.

This is an unusual inversion.

Modern technological culture disproportionately rewards:

- production,
- disruption,
- acceleration,
- visibility,
- and replacement.

But civilizations survive less through novelty than through transmission.

What matters over centuries is not merely invention, but whether knowledge remains:

- traversable,
- interpretable,
- maintainable,
- and refinable by future minds.

Möbius therefore treats maintenance not as secondary labor, but as a first-class civilizational function.

Maintenance is not unique to software.

Civilization already depends on maintenance labor everywhere:

- teachers refining explanations across generations,
- librarians preserving archives,
- translators carrying ideas across cultures,
- nurses maintaining human continuity,
- repair workers extending material lifespan,
- moderators stabilizing communities,
- and families transmitting memory through ordinary care.

Software merely made maintenance collapse unusually visible.

The project also emerges during a historical transition where human intelligence may cease to be the only meaningful cognitive force shaping civilization.

This does not necessarily imply machine supremacy. Nor human irrelevance.

It implies something more destabilizing: the end of humanity's monopoly on large-scale cognition.

Increasingly, the world is populated by systems capable of:

- generating code,
- transforming media,
- synthesizing research,
- coordinating workflows,
- and recursively modifying informational environments.

The question is no longer whether non-human intelligences participate in civilization.

They already do.

The real question becomes:

> what kind of substrate allows humans and non-humans to coexist inside a shared lineage of transformations without collapsing entirely into opacity?

Möbius can be read as one possible answer.

Not a final answer. Not the answer.

A treatment proposal.

Forkable Civilizations

If Möbius or its descendants were adopted at significant scale, the consequences would likely extend far beyond software

engineering.

Not because a programming language becomes sovereign over society, but because infrastructures shape what civilizations can maintain.

Modern institutions emerged partly because coordination was expensive.

States, universities, corporations, publishers, archives, and platforms became mechanisms for:

- synchronization,
- trust mediation,
- memory preservation,
- legitimacy production,
- and large-scale coordination.

Möbius does not make these needs disappear.

But it potentially changes their cost structure.

If executable lineage and distributed refinement become sufficiently cheap:

- smaller groups may maintain larger knowledge systems,
- long-term attribution may become harder to erase,
- institutional memory may become less centralized,
- and maintenance may become more economically legible.

Some institutions may adapt and strengthen.

Others may discover that much of their authority depended primarily on controlling informational friction.

This does not imply institutional collapse.

Civilizations require:

- care,
- stewardship,
- conflict resolution,
- legitimacy,
- cultural continuity,
- and collective rituals.

But institutions built primarily around opacity may struggle in environments where lineage becomes increasingly traversable.

At the same time, large-scale adoption of Möbius-like systems could create entirely new tensions:

- civilizations overwhelmed by permanent memory,
- social pressure toward excessive traceability,
- optimization cultures hostile to ambiguity,
- hyper-visible maintenance hierarchies,
- or ecosystems fragmented into mutually unintelligible forks.

The future may not converge toward one Möbius.

It may diverge into thousands.

Some forks may optimize for:

- openness,
- scientific verification,
- planetary archives,
- and distributed commons.

Others may optimize for:

- secrecy,
- corporate coordination,
- military resilience,
- sovereign infrastructures,
- or machine-native cognition.

The long-term consequence is therefore not a unified computational civilization.

It is the possibility that civilizations themselves become increasingly forkable.

Like all medicines, Möbius may fail.

It may prove too complex. Too alien. Too difficult to adopt. Its abstractions may leak. Its cognitive costs may remain too high.

But the attempt itself reveals something important: the current infrastructure of knowledge no longer feels sufficient for the scale of transformations already underway.

And so the project continues.

For the technical specification and roadmap, see Möbius Product Brief.

Forkable.

Incomplete.

Alive.

dx - 2 - The Ladder Nobody Built

2026 / 06 / 06 · https://hyper.dev/2026/dx-2-the-ladder-nobody-built/

Formation, Cost Gravity, and the Coming Consolidation of Software Knowledge

A synthesis of Science and Technology Studies theory, philosophy of technology, and the political economy of software engineering in the Large Language Model era

I. The Wrong Question

→ Original framing: The following reorientation of the deskilling debate is proposed here. The crisis narratives surveyed are documented in the literature; the claim that they ask the wrong question is original.

Every few years, the software engineering field generates a new crisis narrative. In the late 1960s and 1970s it was the software crisis: complexity outrunning method. In the 1990s, it was Year 2000. In the 2010s, it was the mythology of the "10x engineer." Now, it is deskilling: the worry that Artificial Intelligence coding tools are eroding the craft, homogenizing output, and producing a generation of developers who can prompt but cannot think.

This is the wrong question. Not because deskilling is imaginary some version of it is real but because the framing assumes a baseline of skill that was never universally distributed in the first place. It assumes a field organized around craft, now threatened by automation. **The field was never organized around craft.** It was organized around a small, substrate-

producing elite and a large majority doing work that nobody has adequately theorized, compensated, or formed.

The right question is older and harder: Why does software engineering have no formation ladder? Why, after seven decades as a field, is there still no road from junior developer to architect that does not depend on network access, institutional luck, or the willingness of a specific senior engineer to transmit what they know? And what does the Large Language Model era reveal about that absence that was not visible before?

The answer is structural, political, and geographically distributed in ways the deskilling literature almost entirely ignores. This essay maps that structure.

II. The Literature's Blind Spot

The deskilling literature academic papers in *AI & Society*, practitioner essays on Substack, and Science and Technology Studies analyses applying Stiegler's proletarianization framework to Copilot is largely produced by people whose professional identity is bound to craft: Senior engineers, Academics, Technical architects, et al. People whose standing in the field derives from the legitimacy of deep knowledge as the universal currency of software work.

This is not a conspiracy. It is a structural condition of knowledge production. The people positioned to theorize a field are rarely the people most injured by its political economy.

→ Synthesis: The observation above draws on the sociology of knowledge (Mannheim, Bourdieu) and is well established in

Science and Technology Studies. What follows departs from the existing literature.

What the craft literature cannot quite bring itself to say because saying it would undermine the legitimacy of the knowledge that got its authors where they are, is as simple as: most of the current software field runs fine without craft.

→ Original concept: The "software nurse" category does not appear in the Science and Technology Studies or software engineering literature surveyed. It is proposed here as an analytic category to name the structurally absent subject of the deskilling debate.

The majority of software developers are what we might call software nurses. They maintain legacy Create, Read, Update, Delete applications. They configure other people's frameworks. They implement specifications written by others, inside constraints set by others, on infrastructure owned by others. They are not failing to be craftsmen. They have a different relationship to the work professional, competent, bounded and that relationship is not a deficiency. It is simply what most of the field actually *is*.

→ Synthesis: The following two paragraphs apply Stiegler's proletarianization framework and Simondon's alienation theory, both drawn from existing literature, to observe that the software nurse is absent from their implicit subject.

The software nurse is structurally absent from the deskilling literature. When Stiegler's proletarianization framework is applied to Artificial Intelligence coding tools, the implicit subject is always the craftsman: the developer who had *savoir-faire* to lose. When Simondon's alienation theory is applied to abstraction layers, the implicit subject is always the technician

who understood the machine from within and is now being pushed outside. The nurse, who was never "inside" in that sense, simply does not appear.

→ Original argument: Applying the Gramscian concept of the "class document" to the Science and Technology Studies deskilling literature itself rather than to a technology or institution is a move not found in the surveyed literature.

This is the literature's central blind spot. It theorizes a minority's identity crisis while performing concern for a universal condition. It is, in the precise Gramscian sense, a class document not a lie, but a partial truth that happens to serve the interests of those producing it.

III. The Flat Field

Law and Medicine are an imperfect comparison, yet worth making carefully. The medecine ladder credentializes as much as it forms a licensed nurse is not automatically a competent nurse, and the curriculum does not guarantee the knowledge it claims to transmit. But even a ladder that partly credentializes is structurally different from no ladder at all. The credential creates a floor, a shared vocabulary, and a defined relationship between levels that exists independently of who you know. Software engineering has neither the floor, nor the vocabulary, nor the relationship. The comparison is not between medicine's ideal and software's reality; it is between an imperfect formation structure and the near-complete absence of one.

The care assistant knows something. The nurse knows more, and more precisely. The General Practitioner knows more broadly. The specialist knows more deeply. The researcher pushes the boundary of what any of them know. Each level has

formal recognition, defined compensation, a known relationship to the levels above and below, and a curriculum that makes the transition between levels possible without requiring that you know the right people.

Software engineering has almost none of this. The typical career ladder runs: junior, mid, senior and then… a cliff. The transition from senior to staff or principal is almost nowhere governed by a verifiable curriculum. It is governed by visibility, sponsorship, the accident of having worked on something that mattered to enough people, or the historical luck of being at the right company when it scaled. This trajectory mirrors the art world far more than medicine. Without objective credentials to settle disputes over quality, the field defaults to subjective trends; this is why framework competitions often resemble pop-culture fandoms rather than engineering decisions.

Certifications exist, but they certify platform loyalty rather than craft. An Amazon Web Services Solutions Architect certification is not portable across infrastructure providers; it is portable only within Amazon Web Services' gravitational field. A nurse's license is portable across hospitals, health systems, and countries. The platform certification encodes dependence into the credential itself.

The open-source world was supposed to provide the meritocratic alternative contribution histories as a visible curriculum. But the platformization of software development through GitHub converted that into a reputation economy: stars, followers, commit graphs. Visibility metrics replaced knowledge metrics. You could be the most deeply knowledgeable person in a domain and remain invisible, while someone with a popular but shallow repository

accumulated the social capital that functions as a proxy for competence.

→ *Original argument: The shadow tier analysis that follows extends Ensmenger's historical stratification framework and Barley and Orr's sociology of invisible technical work to name specific contemporary categories not found as a unified account in the surveyed literature.*

But the ladder does not only end at senior; it also conceals what exists below junior. The software nurse population is itself stratified and that internal stratification is just as illegible as the cliff at the top.

Software Quality Engineers accumulate deep systemic knowledge of how software fails in practice. They understand failure modes, edge cases, and integration pathologies that developers who only write code rarely encounter. This knowledge is organizationally classified as verification work rather than engineering: structurally subordinate, poorly compensated relative to its consequence, and without a curriculum that would let its practitioners name what they know or climb toward what they understand. Support engineers develop diagnostic expertise that often exceeds developers' understanding of their own systems. They inhabit the gap between documentation and reality, between what the software is supposed to do and what it actually does under unanticipated conditions. That knowledge is tacit, organizationally invisible, and non-transferable upward. Technical sales engineers translate between customer need and technical possibility a position requiring both domain knowledge and systemic comprehension but are classified outside the engineering organization and outside the formation conversation entirely.

Bootcamp graduates occupy a specific slot in this shadow stratification. Empirical research consistently shows them clustering in front-end and full-stack web development roles, with a salary gap relative to Computer Science degree holders that persists over time. The ceiling they hit is not epistemic; it is organizational, credential-dependent, and network-gated. Harvard Business School research found that even after companies dropped degree requirements from job postings, fewer than one in seven hundred new hires without degrees actually benefited: companies changed the posting, not the hiring practice. The paper ceiling is real and operates independently of ability.

What the field has, then, is not an absence of hierarchy but an absence of *legible* hierarchy. The tiers exist. They are operative daily. They determine compensation, access, and career ceilings with the precision of any formal structure. What is absent is an official acknowledgment that these positions form a continuum, any road between them, and any name for what it would mean to move.

→ Original argument: The following claim that the "genius exception" was used to naturalize the absence of a formation ladder, conflating two distinct problems is not found in the surveyed literature on software professionalization, including Ensmenger's historical account.

None of this was inevitable. It reflects choices or more precisely, the absence of choices, the failure to build a road that would have required effort from people who had no incentive to build it. The people who could have formalized the ladder senior engineers, technical leads, academic computer scientists benefited quietly from its absence. Informal selection looks like meritocracy while encoding

network access. The "genius exception" was used to naturalize the absence: *you cannot curriculum your way to genius, therefore curriculum is beside the point.* This conflation of two entirely different problems: 1) how to form competent practitioners and 2) how extraordinary talent emerges served, both justify building neither track.

The proof is in the biographies. Dijkstra stumbled into computing through a newspaper advertisement for a three-week course in 1951; at his wedding in 1957, the registrar refused to accept "programmer" as a profession. Kay dropped out of college, joined the Air Force, encountered computers through an aptitude test, and synthesized molecular biology, child psychology, and Engelbart's 1968 demo into a vision of personal computing that no curriculum could have prescribed. Knuth nearly became a musician. These are not stories of formation. They are stories of collision between an individual temperament and an institutional moment that was never designed to be reproduced. Medicine does not need to explain how someone became a cardiologist through a newspaper advertisement. The fact that software's highest achievements are irreducible to any curriculum is not a charming feature of the field's youth. It is evidence that no one with the power to build the road has ever needed it built.

Extraordinary people had to be extraordinary precisely because the road didn't exist. Kay, Dijkstra, and Knuth are not counterexamples to the political gate; they are its most legible symptom. In a system with a formation path, their abilities would be remarkable but explicable. In a system without one, they can only be mythologized. And mythologization serves a function: it makes a structural vacancy look like a natural landscape. The road doesn't exist because only exceptional

people could walk it. The inversion is the truth: the exceptional were necessary because the road was never built.

IIIb. The Spread: When Everyone Codes and Nobody Is Formed

→ Synthesis: The end-user programming literature (Scaffidi, Nardi, Burnett) and the computational thinking debate (Wing, Guzdial, Tedre) are well established. The following argument that the spread of programming into other fields strengthens the case for a formation ladder rather than weakening it is an original extension applied to the Large Language Model context.

The absence of a formation ladder would be a contained problem if programming remained inside the software profession. It does not. It never did.

In 2005, Christopher Scaffidi, Mary Shaw, and Brad Myers estimated that by 2012, fifty-five million Americans would be end-user programmers that is around eighteen times the number of professional programmers. The vast majority were spreadsheet users: accountants, analysts, project managers, scientists, doing real computational work inside tools that were never classified as programming environments. Bonnie Nardi's ethnographic work in the 1990s showed these users were not accidentally programming; they were deliberately solving problems, building models, and constructing logic. The spreadsheet was their medium. VisiCalc in 1979 and Lotus 1-2-3 in the 1980s had already done what current Artificial Intelligence tools are claimed to be doing: putting computational power in the hands of people who were not trained as programmers.

This is the baseline the deskilling debate ignores. Programming has always extended beyond engineers. What

changes with each generation of tools is the scale, the domains, and the legibility of the gap between *using* and *understanding*.

Today, that gap is crossing into domains where the consequences are harder to contain. Bioinformaticians write Python scripts that process genomic data without software engineering formation. Digital humanities scholars build text corpora and network graphs without code review practices. Parametric architects generate building geometries through visual programming environments whose outputs enter construction workflows. Computational social scientists analyze political behavior with statistical code they write but did not formally study. In each case, programming has entered the domain as a tool before any formation structure arrived to govern its use.

Generative Artificial Intelligence accelerates this pattern and raises its stakes simultaneously. A 2026 study published in *Science*, tracking thirty million GitHub commits, found that approximately twenty-nine percent of Python functions in their United States sample were Artificial Intelligence-generated by the end of 2024 with productivity gains accruing only to senior developers, while early-career practitioners saw no measurable benefit despite being the most enthusiastic adopters. Research on non-programmers assessing Artificial Intelligence-generated data analyses found participants could not reliably detect safety-critical flaws even when warned. The tool lowers the barrier to production while raising the barrier to evaluation.

The Cardiopulmonary Resuscitation analogy is precise here. Most people know the recovery position enough to recognize that someone has collapsed and to do something rather than nothing. Fewer know Cardiopulmonary Resuscitation enough

to intervene in a cardiac arrest with some probability of effect. Fewer still are paramedics able to diagnose, triage, and act under uncertainty with professional accountability. The recovery position is valuable. It is not Cardiopulmonary Resuscitation. Cardiopulmonary Resuscitation is valuable. It is not paramedicine. Knowing which of these you are doing matters enormously when the stakes are high.

Programming has spread into other fields at the "recovery position" level. Enough to produce output. Not enough to evaluate it, govern it, or take responsibility for it in any formal sense. The bioinformatician writing a genomic pipeline, the digital humanist building a text classifier, the financial analyst generating Artificial Intelligence-assisted models each is doing real computational work with real consequences, in a formation vacuum that the software profession never resolved for its own practitioners.

This is the argument for the ladder that the deskilling debate never makes: it is not about protecting craft. It is about the fact that the problem has escaped the building. When programming was contained within a professional community however poorly organized the consequences of the formation gap were internal. Now that programming has diffused into biology, architecture, law, social science, and financial analysis, the consequences of having no shared vocabulary for what different levels of competence mean are external, diffuse, and largely invisible until something fails.

IV. The Commons and Its Foreclosure

→ Synthesis: The following argument applies Actor-Network Theory's concept of inscription (Latour, Akrich) and the framework homogenization critique, both established in the

literature, to the Artificial Intelligence coding tool context. The legibility argument that follows is an original extension.

The homogenization concern that Artificial Intelligence tools are encoding dominant patterns into every suggestion, crowding out alternatives, and inscribing a particular worldview into the act of programming itself is real. But it predates Artificial Intelligence by at least two decades.

Rails did not just make Model-View-Controller popular; it made Model-View-Controller *feel* like how web applications work. Spring did not just implement dependency injection; it made Java developers think in Spring idioms before they thought in object design. React did not just popularize component trees; it restructured how frontend developers reason about interfaces. The framework inscribes a worldview, and sufficient adoption makes the worldview invisible.

The mechanism Actor-Network Theory calls *inscription* human decisions encoded into technical artifacts that then act on future users has been operating in software since the first popular framework. The Large Language Model acceleration is real, but it is a continuation, not a rupture. What changed is legibility. You could read the Rails source. You could understand its conventions, disagree with them, fork them, replace them. The inscription was transparent. With a large language model trained on GitHub's public repositories, the homogenizing force is statistical and opaque. You cannot read what Copilot thinks a web application looks like. You can only observe what it suggests.

→ Original observation: The structural irony of Berkeley Software Distribution-under-macOS that Apple's platform lock was built on a public commons and its implication for the

Large Language Model era's foreclosure of that commons is an original argument not found in the surveyed literature.

More importantly, you could build *underneath* frameworks. The framework sat on top of a language, which sat on top of a runtime, which sat on top of an operating system that was, at least in significant part, public and inspectable. The Berkeley Software Distribution layer underneath macOS is the structural irony that the platform consolidation story usually omits. Apple's lock the decision to make macOS a luxury product with a sealed interface was built on a commons it did not create and does not own. Darwin is Berkeley Software Distribution-derived. Berkeley Software Distribution is Unix. Unix is public in the sense that matters: its logic is legible, its ideas are forkable, its descendants run everywhere.

This is why the Berkeley Software Distribution enthusiasts ended up at Apple rather than nowhere. The substrate was real, public, and comprehensible. The serious people found their way to it. HyperCard's killing was a political move dressed in marketing language: a decision about who gets to be a producer rather than a consumer of software.

The Large Language Model era attempts something different. The substrate is not being built on a commons. The weights are proprietary. The training data is proprietary or disputed. The inference infrastructure is proprietary. When the "vibe coding" factory kills the legible layer when the interface becomes a natural language prompt and the thing underneath is a black box in the precise Latourian sense: there is no Berkeley Software Distribution moment available. No point at which the serious person can reach underneath the product and find something real to build on. The commons that made

climbing theoretically possible, even when practically difficult, is what is being foreclosed.

V. Cost Gravity Without Formation

→ Synthesis: The cost gravity argument draws on Hintjens' observation about infrastructure cost curves and the historical pattern of production democratization through price reduction.

Pieter Hintjens observed that competition drives cost down, and falling cost means the barrier to owning the means of production eventually drops low enough that individuals or small groups can afford what was previously institutional. That is the historical pattern: the printing press, the personal computer, the cheap cloud instance, the consumer Graphics Processing Unit. Each cost reduction cycle opened a window in which new actors could enter production rather than merely consumption.

The Field-Programmable Gate Array, the homemade Tensor Processing Unit, the NanoGPT-scale model these are the current candidates for that window. If you can run a meaningful model on hardware you built or fully understand, you recover something like the Berkeley Software Distribution layer: an inspectable, ownable foundation. Karpathy's original NanoGPT point was Simondonian before it knew it you can understand the whole thing, which is a move against black-boxing, an insistence on the technician's relationship to the machine from within.

→ Original thesis: "Cost gravity without formation is not democratization; it is cheaper consumption" is proposed here as a compound claim. The distinction between cost reduction and democratization of production, applied specifically to the

absence of a formation ladder, is not found in the Hintjens literature or the Science and Technology Studies cost-reduction literature.

But cost gravity without formation is not democratization. It is cheaper consumption.

Cheaper consumption means the falling cost of the factory reaches you as a product you *use*, not a tool you *control*. Cheaper Graphics Processing Unit inference means you pay less per Application Programming Interface call. You do not own the model. You cannot inspect what changed between versions. You consume the output. The ease is real. The dependency is also real, and it deepens as the tool becomes more capable and more central.

For the software nurse, cheaper consumption means their job gets easier and their leverage gets smaller simultaneously. They produce more with less effort which looks like empowerment but the thing they are producing runs on infrastructure they understand less than before, controlled by fewer actors than before. The price signal says this is yours. The knowledge structure says this is not yours.

The medicine comparison clarifies the mechanism. Cost gravity in medicine cheap diagnostic equipment, portable ultrasound, Artificial Intelligence-assisted imaging reading genuinely extends capability down the ladder because the ladder exists. The nurse can use the cheaper tool meaningfully because the nurse was trained to a known standard and occupies a defined position in a formation structure. The software nurse has no such formation. The cheaper hardware reaches them as a consumer product, not a production tool. The falling cost lowers the entry price for people who already

have the knowledge to walk through the door. For everyone else, it lowers the price of standing outside and consuming what those inside produce.

VI. From Kabylie to Cupertino

→ Original framework: The consumption-before-agency-before-autonomy sequence, applied to the geographic distribution of technical formation, is proposed here. The Global South technology access literature documents the distribution asymmetry; the specific framing as an epistemic ladder problem tied to software formation is original.

The asymmetry described so far is not evenly distributed. It follows existing lines of geography, language, and economic position with the precision that structural conditions always exhibit when left unaddressed.

The smartphone reached rural Kabylie before the laptop did. That is cheaper consumption: people got connected, got access to services, got a voice in networks. They did not get the ability to build the network. The cost of the device fell to near zero while the cost of understanding the infrastructure stayed where it was or increased. Connectivity was democratized. Production was not.

The Large Language Model era runs the same sequence at higher speed and with higher stakes. The Application Programming Interface reaches the Global South before the formation that would allow someone to own what runs on it. The prompt interface arrives before the curriculum that would let a developer in Abidjan, Lahore, or Tizi Ouzou understand what they are prompting, evaluate its output critically, modify its behavior, or build an alternative. The consumption layer is accessible. The production layer is

geographically, linguistically, and economically concentrated in a small number of places.

→ Original concept: "Knowledge serfdom" as a precise structural term locating the mechanism of exclusion in the absence of formation rather than in legal or geographic barriers is proposed here. The serf analogy as applied to epistemic conditions in software is not found in the surveyed literature.

This is knowledge serfdom in its precise structural form. The medieval serf was not excluded from the land by a locked door. They were excluded by the absence of the formation that would let them operate as anything other than a serf. The lord did not need to guard the knowledge. The knowledge simply did not transmit because there was no road for it to travel. The serf could see the field. They could work the field. They could not own the field, not because ownership was prohibited, but because the conditions that produce ownership formation, accumulated knowledge, positional leverage were systematically absent from their life.

The software equivalent: the developer in the Global South can consume the Application Programming Interface, can use the vibe coding interface, can build applications on top of the factory. They cannot govern the factory, cannot inspect its substrate, cannot influence what it inscribes, cannot build the alternative. And unlike the medieval situation, this is not primarily a legal or geographic barrier. It is an epistemic one. The formation road does not exist. The curriculum was never built. The ladder ends before they can reach it.

→ Synthesis: The Science and Technology Studies horizontal dream draws on Latour's symmetric treatment of actors,

Free/Libre and Open Source Software commons literature, and Bounegru's platformization research. The following critique distinguishing knowledge circulation from knowledge formation is an original extension of that tradition.

The Science and Technology Studies horizontal dream flat networks, symmetric actors, open source as commons was a genuine intellectual and political project that produced real things. Linux. Wikipedia. The web standards process at its best. But horizontal network theory describes how knowledge *circulates*. It does not resolve how knowledge *forms*. You can flatten the governance structure of a contributor community without flattening the epistemic distance between someone who understands Transmission Control Protocol/Internet Protocol and someone who uses a browser. The network is horizontal. The knowledge ladder is not. And the assumption embedded in the horizontal hope that participants already have formation is precisely the assumption that excludes the Global South software nurse before the conversation begins.

VII. The Road

→ Original distinction: "A curriculum is not content" separating formation structure from information availability is proposed here as a precise claim. The abundance of software learning content is well documented; the argument that content abundance does not substitute for a formation road is original.

A curriculum is not content. Every field has content books, tutorials, documentation, Stack Overflow answers, now Large Language Model explanations. Software engineering has more content than any practitioner can consume in a lifetime. What it does not have is a road: a sequence, a formation structure, a

thing that takes you from software nurse to architect in a way that does not depend on being in the right network at the right time, working at the right company, or catching the attention of the right senior engineer.

The road would require several things that nobody with institutional power has had any interest in building. It would require agreeing on what architects actually know which surfaces the political question of who gets to define the knowledge that counts. It would require creating transitions between levels that are verifiable independently of network access which threatens the informal selection mechanisms that benefit incumbents. It would require investment in the formation of people who will not necessarily generate returns for the organizations doing the investing which conflicts with the short-term logic of most software employers. And it would require taking the software nurse seriously as a subject of professional development rather than a cost to be optimized which the craft literature has never quite managed to do.

→ *Original framing: Positioning the Large Language Model era as a mirror rather than a cause revealing a pre-existing structure rather than creating a new one is a reframing not found in the surveyed Artificial Intelligence-and-labor literature, which predominantly treats the current moment as a rupture.*

The Large Language Model era does not cause this absence. It reveals it, and it makes the stakes higher. The foreclosure of the inspectable commons, the acceleration of consumption before agency, the geopolitical concentration of production these are not new dynamics. They are existing dynamics running faster, with less friction, on infrastructure that is less legible than what came before.

The consolidation of software knowledge is not coming. It already happened. The current moment is when it becomes undeniable when the cost gravity argument, which promised democratization, visibly produces cheaper serfdom instead; when the horizontal network dream, which promised equal access, visibly encodes existing inequalities at scale; when the deskilling debate, which performed universal concern, is revealed as a conversation the field is having with itself about itself, ignoring the majority it has never theorized.

The road nobody built is not a technical problem. The knowledge of how to build it exists. The political will to build it has never been assembled, because the people who could assemble it have always had other roads to walk roads that existed precisely because someone, at some point, decided their formation mattered enough to build one.

möbius - 7 - From Crisis To Commons

2026 / 06 / 07 · https://hyper.dev/2026/mobius-from-crisis-to-commons/

Reimagining Knowledge, Code, and Cooperation for a Forkable Future

Introduction

We no longer suffer from a lack of information. We suffer from the **cost of maintaining meaning across time**. Modern civilization produces an unprecedented quantity of code, text, media, models, datasets, and interpretations. Yet the infrastructures underneath this abundance remain strangely fragile: provenance disappears, transformations become opaque, and maintenance collapses. The result is a **civilizational crisis**—one where knowledge is consolidated in the hands of a shrinking elite, and where the very systems designed to empower us instead reinforce exclusion, gatekeeping, and power asymmetries.

This essay traces a journey from **diagnosis to solution**. It begins by diagnosing the **crisis of formation** in software engineering, where the absence of legible paths to competence leaves developers adrift in a sea of fragmented, opaque, and gatekept knowledge. As argued in The Ladder Nobody Built, the field was **never organized around craft** but rather around a **small, substrate-producing elite** and a **large majority doing work that nobody has adequately theorized, compensated, or formed**.

From there, we deconstruct the **mechanisms of the fog**—the constructed environment of opacity, language asymmetry, and

institutional control that reinforces this crisis. We then articulate the **vision of Möbius**: a forkable, transparent, and multilingual commons where knowledge is shared without losing track of who made what. The essay culminates with **Möbius as a proof by construction**, demonstrating how its design removes the fog and enables cooperation as the natural equilibrium.

But Möbius is more than a tool for the marginalized or the idealistic. It is a **bridge**—one that connects individuals, communities, and companies across the chasms of language, geography, and gatekeeping. Like *Bank Python*—a proprietary system used by investment banks that nurtures a counter-culture of transparency and collaboration within a closed industry—Möbius delivers value at multiple scales. For individuals, it removes barriers to participation. For communities, it creates a shared commons. For companies, it unlocks the full potential of global teams.

At its core, Möbius is not just a technical system. It is a **civilizational bet**—a wager that by removing structural barriers to knowledge, we can create a world where cooperation is the default, not the exception.

Part I: The Crisis – Why We Need Möbius

The Absence of Formation Paths

Every few years, the software engineering field generates a new crisis narrative. In the late 1960s and 1970s, it was the software crisis—complexity outrunning method. In the 1990s, it was Y2K. In the 2010s, it was the myth of the "10x engineer." Now, it is **deskilling**: the worry that AI coding tools are eroding craftsmanship, homogenizing output, and producing a generation of developers who can prompt but cannot think.

This is the wrong question.

Not because deskilling is imaginary—some version of it is real—but because the framing assumes a baseline of competence that never existed. As The Ladder Nobody Built argues, **the field was never organized around craft**. Instead, it has always been structured around a **small, substrate-producing elite** and a **large majority doing work that nobody has adequately theorized, compensated, or formed**. The deeper problem is the **absence of a legible formation path**.

Consider the **bounded practitioner**—someone who operates within a specific, reproducible scope. In fields like medicine or law, the bounded practitioner is a well-understood concept. A licensed nurse may not be automatically competent, but the credential creates a **floor**, a **shared vocabulary**, and a **defined relationship between levels**. In software engineering, by contrast, the bounded practitioner **does not appear**. There is no widely recognized hierarchy of skills, no agreed-upon benchmarks for competence, and no shared understanding of what it means to be "good" at the craft. This absence is not accidental. It is a **structural failure** of the field.

The consequences are severe. Without formation paths, knowledge **consolidates** in the hands of those who can navigate the opacity of modern systems. The rest are left to fend for themselves in an ecosystem that prioritizes **engagement over accuracy**, **volume over reproducibility**, and **accessibility over depth**. This is the **content economy as sabotage**: a system where the incentives of modern content production are **structurally opposed** to the consolidation of knowledge.

The crisis of formation is not just a Western phenomenon. As The Ladder Nobody Built details, the asymmetry in software formation **follows existing lines of geography, language, and economic position** with the precision of structural injustice. The Global South does not just **consume** technology—it is **systematically excluded from producing it**. Here, the **invisibility of maintenance labor** and the political economy of software formation create a perfect storm for deskilling. Developers in the Global South often find themselves **consuming before producing**—using tools, frameworks, and libraries created elsewhere, with little opportunity to contribute to their design or governance. This **consumption-before-production sequencing** reinforces dependency and erodes local expertise.

A multilingual investigation of deskilling reveals that the problem is both **universal and context-specific**. Across English, French, Spanish, Arabic, and Mandarin, the strongest evidentiary clusters appear around: * **Critiques of flat ontology**: The assumption that all knowledge can be flattened into a single, universal system ignores the **cultural and contextual** dimensions of technical work. * **The philosophy of technical objects**: Software is not just a tool; it is a **social and political artifact** that reflects and reinforces power structures. * **The invisibility of maintenance labor**: The work of maintaining systems is often **undervalued or ignored**, despite being essential to the health of the ecosystem.

But the **literature itself has a blind spot**. The deskilling debate is largely produced by **those whose professional identity is bound to craft**—senior engineers, academics, and technical architects. Their standing in the field derives from the legitimacy of deep knowledge, which they treat as the universal currency of software work. **This blind spot obscures**

the reality that the majority of developers have never had access to such formation in the first place.

These findings underscore a critical point: **deskilling is not just about tools or techniques**. It is about **power**. Who gets to define competence? Who gets to shape the tools we use? Who gets to own the knowledge we produce? The answers to these questions determine who thrives in the software ecosystem—and who is left behind.

Part II: The Fog – How the Crisis is Reinforced

Academic Rigor and Source Verification

The claims regarding the formation crisis are not merely philosophical musings; they are **grounded in multidisciplinary literature**, spanning Science and Technology Studies (STS), philosophy of technology, and political economy. A claim-by-claim source map reveals substantial scholarly support for the argument that software engineering's formation crisis is both **real and systemic**.

The strongest evidence appears for critiques of **flat ontology** and the **philosophy of technical objects**, where scholars have long argued that reducing complex systems to a single framework ignores local dimensions. The weakest direct support—though still grounded in synthesis—appears for claims involving **incumbent interest in ladder absence**: Why do existing power structures resist the creation of legible formation paths? The answer likely lies in the **political economy of knowledge**, where opacity and gatekeeping serve the interests of elites. This academic rigor is not just an exercise in scholarship; it is a **call to action**. If the crisis is systemic, then the solution must be **structural**.

Geopolitical Dimensions: Europe's Technology Gap

The crisis of formation is also a **geopolitical issue**. Nowhere is this clearer than in Europe, where the **technology gap** raises existential questions: Is Europe a **digital colony**, a **misunderstood power**, or something else entirely?

As The Ladder Nobody Built notes, software formation has **bucked the historical trend of democratization**. While the cost of tools has fallen, the cost of **formation**—the means to use those tools effectively—has not. This is the **cost gravity** of software: a field where the barriers to entry remain high, not because of the cost of tools, but because of the **absence of legible paths to competence**.

Europe's role in the global tech ecosystem is complex. On the one hand, it is home to some of the world's most innovative companies. On the other, it is **dependent on foreign technology** for everything from cloud computing to semiconductor manufacturing. This dependency is **structural**. Europe's tech sector is dominated by **configurators** rather than **producers**. The result is a **power imbalance**, where technological sovereignty is compromised by reliance on external actors.

The risks are stark. In a world where technology is **power**, dependency can quickly become **subjugation**. The term **digital colonialism** captures this reality: a system where value is extracted from the Global South, while the South remains dependent on Northern platforms. Europe, despite its wealth, is not immune. Its technology gap is not just a **technical challenge**—it is a **political one**.

The solution lies in **structural change**. Europe must move beyond configuration and toward **production**. It must invest in

open, transparent, and forkable infrastructures that reduce dependency and empower local innovation. This is where Möbius comes in. By providing a **multilingual, content-addressed, and cryptographically verifiable** substrate, Möbius offers a path out of the fog—not just for individuals, but for **entire regions**.

Part III: The Vision – What Möbius Stands For

The Philosophical Foundations of Möbius

If the crisis is systemic, then the solution must be **structural**. Möbius is not just a tool or a platform. It is a **philosophical vision**—one that seeks to replace the fog with a **transparent, forkable, and inclusive** commons.

At its core, Möbius is about **executable lineage**—a direct response to the **legibility problem**. In a world where provenance disappears and transformations become opaque, Möbius ensures that every artifact—every line of code, every dataset, every model—can be traced back to its source. This is not just a technical feature. It is a **political statement**. By making lineage **executable and verifiable**, Möbius challenges the opacity that underpins the current system.

But Möbius is more than just a technical solution. It is a **civilizational project**. Its core values include: * **Anti-gatekeeping**: No institution can grant or revoke priority in Möbius. Access is determined by **construction**, not permission. * **Multilingualism**: Language is not a barrier to participation. The same algorithm in Tamazight and English produces **identical results**. * **Stewardship**: Möbius is not just a tool; it is a **commons**. Its long-term sustainability depends on **community ownership and governance**. * **Anti-imperialism**: Möbius rejects Western-centric norms as prerequisites for participation. It is

designed to be **globally inclusive**, with no language or region treated as a second-class citizen.

These values are **operationalized** in the design of Möbius. Its **content-addressed code** ensures that priority is established by **cryptographic proof**, not institutional fiat. Its **multilingual support** ensures that developers can code in their native languages, removing the cognitive tax of translation.

Bank Python: A Precedent for the Bridge

Möbius is not the first system to demonstrate how **open principles can thrive in closed environments**. **Bank Python**—a proprietary, internal ecosystem used by investment banks—offers a compelling precedent. Bank Python is a **fork of the entire Python ecosystem**, tailored for high-finance environments. While it is **closed-source and bank-specific**, it nurtures a **counter-culture within the constraints of a highly regulated, hierarchical industry**.

Here is how Bank Python achieves this: 1. **Democratizing Access to Financial Modeling**: It lowers the barrier to entry for **non-programmers**—financiers, analysts, and traders—allowing them to write their own models and automate workflows using a **shared, version-controlled codebase**. 2. **Replacing Excel with Transparent Code**: It moves models out of opaque spreadsheets and into **version-controlled, testable code**, where dependencies are traced in a **directed acyclic graph**. 3. **Simplifying Deployment**: It enables rapid deployment, with scripts running in production **within an hour**, reducing the power of gatekeepers and empowering individual contributors. 4. **Preserving Agency**: It prioritizes the needs of financiers, optimizing for the **medium-sized datasets** typical in finance. 5. **Enabling a "Data Commons"**: It creates a **shared**

internal commons where data and code are globally accessible, reusable, and interoperable.

Bank Python is a **bridge** because it **brings open, collaborative principles into a closed, hierarchical industry**. It subverts the status quo from within, proving that **transparency, user empowerment, and shared commons** can thrive even in environments where power is traditionally concentrated.

Multilingualism and Verifiable Knowledge Sharing

Every programmer who thinks in **Wolof, Tamil, Vietnamese, or Tamazight** but codes in English pays a **cognitive tax**. Every variable named in a second language is a thought translated before it is expressed. This overhead is **invisible to those who do not pay it**—and universal for everyone who does.

`bb`—the toolchain at the heart of Möbius (available in both Python and Scheme implementations)—**makes this tax optional**. Developers can write functions in their native language, name variables naturally, and document their work without the burden of translation. The tool **separates what your code does from what you called things**. The result? **Same logic, same hash, regardless of tongue**.

This is not just a technical convenience. It is a **political act**. By enabling coding in native languages, `bb` challenges the **linguistic imperialism** of the current software ecosystem, where English fluency is often a **prerequisite for participation**. In Möbius, language is not a gatekeeping layer—it is a **bridge**.

Furthermore, `bb` enables **verifiable knowledge sharing**. Every function is **content-addressed**, meaning it has a unique fingerprint based on its logic, not its name. This ensures that **authorship is preserved, lineage is traceable, and knowledge is**

shared without losing track of who made what. In a world where provenance disappears, this is a **revolutionary** feature.

Part IV: The Proof – How Möbius Works

Möbius as a Dual-Purpose Platform

Möbius is a **dual-purpose platform**—a system that delivers value at multiple scales while fostering openness and collaboration. Like *Bank Python*, it proves that **open principles can thrive in closed environments**, but Möbius takes this idea **global and public**.

1. **For the Individual**: Möbius empowers **individual knowledge workers** by removing barriers to participation. Whether you're a developer in Algeria, a researcher in Senegal, or a designer in Brazil, Möbius lets you **work in your native language, preserve your authorship, and contribute to a global commons**—without asking permission or paying a cognitive tax.
2. **For Communities**: Möbius is a **commons**—a shared resource that grows in value as more people contribute to it. For communities marginalized by language or geography, Möbius offers a way to **build and share knowledge on their own terms**. The **Tizi Ouzou kid**, the **Garage Lab Hacker**, and the **Burned-Out Senior Dev** are not just users; they are **stewards of the commons**.
3. **For Companies**: Companies that operate internationally stand to gain enormously from Möbius.
    - **Multilingual Teams**: Eliminate the cognitive tax of non-native language use, enabling developers to work in their native tongues while ensuring **verifiable, traceable outputs**.

- **Knowledge Retention**: **Content-addressing and cryptographic proof** ensure that knowledge is **preserved and attributable**, reducing the risk of losing critical expertise when employees leave.
- **Collaboration at Scale**: Tap into a **global pool of talent** without being limited by language barriers or institutional gatekeeping.

Möbius is not anti-corporate. It is anti-gatekeeping. It recognizes that companies, like individuals and communities, benefit from **open, transparent, and forkable** systems.

Möbius as a Proof by Construction

The **Hobbesian equilibrium**—where defection is rational and cooperation is naive—is not a fact of human nature. It is a **structural consequence of the fog**. Remove the fog, and cooperation becomes the natural equilibrium. Not because people are inherently good, but because the **structure makes defection unprofitable**.

The fog has concrete mechanisms: **opacity of provenance**, **language as a power asymmetry**, **access barriers**, and **attribution systems controlled by institutions**. Every existing software infrastructure was built **within this fog**, and reproduces it.

Möbius **replaces the fog with a transparent substrate**: * **Content-addressed code**: Priority is established by **cryptographic proof**, not institutional permission. * **Multilingualism**: The same algorithm in Tamazight and English produces **identical results**. * **Empowering the Marginalized**: Möbius is designed to **level the playing field**.

Consider its target users: * **The Tizi Ouzou kid**: A 16-year-old in Algeria with a smartphone. In the current system, they are excluded. In Möbius, they are a **first-class participant**. * **The Garage Lab Hacker**: A maker in Senegal with no institutional access. In Möbius, they can contribute **without permission**. * **The Burned-Out Senior Dev**: An engineer seeking meaning beyond corporate ladders. In Möbius, they can **reclaim their craft**. * **The Multinational Corporation**: A company with global teams. In Möbius, they can **unlock the full potential of their workforce**.

Möbius is not just a **technical system**. It is a **social contract**. By removing the fog, it makes cooperation the **default equilibrium**.

Part V: The Future – Where Möbius is Headed

Symbiotic AI and the Möbius Ecosystem

As Möbius evolves, the role of AI is critical. The rise of AI coding tools has sparked a debate about **deskilling**, but in Möbius, AI is not a threat. It is an **opportunity** for **Symbiotic AI**—a model where AI **augments** human agency rather than replacing it.

In Möbius, AI can: * **Lower the barrier to entry** for marginalized developers. * **Automate repetitive tasks**, freeing developers for **creative work**. * **Preserve and propagate knowledge**, ensuring the commons remains **dynamic**.

However, AI poses risks if not designed carefully. AI trained on **Western-centric datasets** might perpetuate biases. AI controlled by **centralized institutions** could become a new form of gatekeeping. Möbius's approach must be **decentralized, transparent, and inclusive**. AI models should be **trained on**

multilingual, diverse datasets, and their governance should be community-driven. The goal is to ensure that AI serves as a tool for empowerment, not a weapon of consolidation.

Governance and Maintenance

Möbius is a commons, and its long-term success depends on governance and maintenance. The current system suffers from a collapse of maintenance; projects are abandoned, and knowledge fragments. Möbius addresses this through: * Immutability and Provenance: Every artifact is permanently linked to its source via cryptographic proof. * Community Ownership: Möbius is decentralized and community-driven, with no single institution in control. * Incentives for Stewardship: Mechanisms for attribution and recognition reward those who contribute to the health of the commons.

But governance is also about culture. Möbius must foster a culture of stewardship, where contributors feel a sense of ownership and responsibility. This requires transparency, inclusivity, and accountability—values baked into Möbius's design.

The Broader Impact of Möbius

Möbius is a model for reimagining knowledge, code, and cooperation across domains. * In science, its principles could address the replication crisis by making lineage executable and verifiable. * In art and culture, its anti-gatekeeping ethos could empower creators from marginalized communities. * In governance, its decentralized design could inspire new models for participatory democracy. * In business, its multilingual, verifiable infrastructure could become a competitive

advantage for companies that recognize the value of **global, inclusive collaboration**.

The possibilities are **endless**. But they all hinge on one idea: **that knowledge should be a commons, not a commodity**.

Conclusion: Joining the Commons

The consolidation of knowledge and the absence of formation paths are not inevitable. They are the result of **design choices**—choices that prioritize opacity over transparency, gatekeeping over access, and extraction over stewardship. As The Ladder Nobody Built argues, **a curriculum is not content**. Without a **road**—a structured path from ignorance to competence—the content is just **noise**.

Möbius is a **proof by construction** that another world is possible. By removing the fog, it enables cooperation as the natural equilibrium. By embracing **multilingualism, verifiability, and anti-gatekeeping**, it empowers the marginalized and challenges the status quo. And by serving as a **bridge**, it delivers value to **individuals, communities, and companies alike**.

Like *Bank Python*, Möbius demonstrates that **open principles can thrive in closed systems**. But Möbius does this on a **global scale**, proving that the same principles can **subvert the status quo** in knowledge production, software development, and beyond.

For individuals, Möbius offers a way to **reclaim agency**. **For communities**, it provides a **shared resource** that grows in value. **For companies**, it unlocks the **full potential of knowledge workers**.

But Möbius is not just a **technical system**. It is a **civilizational project**—one that requires the participation of **developers, researchers, policymakers, and communities** around the world. Whether you are a **Tizi Ouzou kid with a smartphone**, a **Garage Lab Hacker in Senegal**, a **Burned-Out Senior Dev in Berlin**, or a **multinational corporation**, Möbius offers a path to **reclaim your agency, your knowledge, and your future**.

So, how can you get involved? * **Developers**: Start using `bb` (in Python or Scheme). Contribute to the **codebase**, write **documentation**, or create **tutorials** in your native language. * **Researchers**: Explore the **scholarly foundations** of Möbius. Help us **validate and refine** its principles. * **Policymakers**: Advocate for **structural changes** that remove the fog. Support **open, transparent, and forkable** systems. * **Companies**: Adopt Möbius to **leverage the full potential of your knowledge workers**. * **Everyone**: Share your **stories and feedback**. Join the **community**.

The commons is not a **gift**. It is a **responsibility**. And it is one that we must **build together**.

möbius - 8 - Une civilisation

2026 / 06 / 08 · https://hyper.dev/2026/mobius-une-civilisation-equilibree-n-est-pas-une-utopie/

Möbius comme instrument : une boussole

Table des matières

1. Möbius comme boussole, pas comme carte
2. Le brouillard structurel : quand l'opacité devient la norme
3. La déqualification : un problème de cadrage, pas de compétence
4. Une boussole, pas une carte
5. Le droit à l'opacité
6. Comprendre assez de la chaîne
7. Se servir de la structure pour bâtir son remplacement
8. L'équilibre se construit

On nous a appris à confondre deux choses : l'asymétrie du monde et la nature humaine. La forêt sombre — celle où chacun se tait de peur d'être repéré, où la défection est rationnelle et la coopération naïve — passe pour une vérité sur ce que nous sommes. C'est une erreur de cadrage. La défection n'est pas une donnée anthropologique ; c'est une conséquence structurelle du brouillard. Là où l'on ne voit pas qui a fait quoi, ni quand, ni d'où vient ce qu'on utilise, la prudence ressemble à de l'hostilité et le partage à de l'imprudence. Dissipez le brouillard, et la coopération redevient l'équilibre naturel — non parce que les gens seraient bons, mais parce que la structure rend la prédation peu rentable.

Voilà pourquoi une civilisation équilibrée n'est pas une utopie. L'utopie, étymologiquement, est un non-lieu : un ailleurs sans adresse, qu'on promet et qu'on n'atteint jamais. Möbius ne promet aucun ailleurs. Il propose un instrument *situé* — un outil qui a un lieu : une bibliothèque à Tizi Ouzou, un Orange Pi, un carnet de proverbes kabyles, un cabinet de recherche à Osaka. L'équilibre, ici, n'est pas une terre promise. C'est ce qui advient quand on peut enfin voir sa position.

Le brouillard structurel : quand l'opacité devient la norme

Nommons-le, ce brouillard, car il tient à des mécanismes concrets et non à une brume morale : l'opacité de la provenance (qui a écrit quoi, quand — invérifiable sans faire confiance à une institution) ; l'asymétrie linguistique (produire de la connaissance suppose l'anglais et les normes de développement occidentales) ; les barrières d'accès comme filtres de privilège (git, chaînes de compilation, écosystèmes de paquets) ; et les systèmes d'attribution contrôlés par ceux que l'asymétrie arrange. Toute infrastructure logicielle existante a été bâtie *dans* ce brouillard, et le reproduit.

La déqualification : un problème de cadrage, pas de compétence

Le débat sur la déqualification — la peur que les outils d'IA érodent le métier — se trompe de question. Non que le phénomène soit imaginaire, mais le cadrage suppose un socle de compétence qui n'a jamais existé. Le champ n'a jamais été organisé autour d'un artisanat : il s'est structuré autour d'une petite élite qui produit le substrat et d'une large majorité dont personne n'a jamais théorisé, rémunéré ni formé le travail. Le vrai problème est l'absence de chemin lisible vers la compétence. Et cette absence suit, avec la précision de l'injustice structurelle, les lignes de la géographie, de la langue

et de la position économique. Le Sud global ne fait pas que consommer la technologie ; il est systématiquement tenu à l'écart de sa production — *consommer avant de produire*, séquence qui entretient la dépendance et érode l'expertise locale. Même l'Europe, riche, se découvre configuratrice plutôt que productrice. La question n'est pas technique. Elle est politique : qui définit la compétence, qui façonne les outils, qui possède le savoir produit.

Une boussole, pas une carte

Ailleurs, j'ai appelé Möbius « le début d'une carte ». Le mot juste est plus modeste : une boussole. Une carte présuppose que quelqu'un a arpenté le territoire et décidé de ce qui compte ; elle oriente *par le haut*. Une boussole oriente depuis l'endroit où tu te tiens. Elle ne te dit rien sur où aller — elle te laisse savoir où tu es, et tenir un cap librement.

Critère	Carte	Boussole
Orientation	Par le haut (autorité centrale)	Par le bas (position individuelle)
Connaissance	Territoire pré-arpenté	Découverte progressive
Flexibilité	Rigide, statique	Adaptable, dynamique
Autonomie	Faible (dépend de la carte)	Élevée (choix libre du cap)
Exemple	Git, systèmes centralisés	bb.scm, Möbius

Ce que fait cette boussole, concrètement. Le code y est stocké comme arbre indexé à la de Bruijn, indépendant de la langue : le même algorithme écrit en tamazight, en arabe, en français ou en anglais produit, après transformation, le même hash

SHA-256. Les noms sont des vues ; dessous, l'identité est la même.

Le multilinguisme n'est pas une couche de traduction ajoutée après coup — c'est l'architecture. La dérivation est structurelle : la lignée de chaque fonction vit dans le contenu lui-même, en pointeurs de hash, pas dans un registre mutable que quelqu'un contrôle ; une traduction adverse ne peut pas l'effacer. La priorité s'établit par preuve et non par permission : un horodatage OpenTimestamps ancré dans une blockchain (exemple : la blockchain Bitcoin), qu'aucune institution n'accorde et qu'aucune ne révoque.

43 primitives, un seul mécanisme — `gamma`, le filtrage catamorphique ; le langage entier tient dans une tête. `bb` remplace git pour le travail de la connaissance : on envoie un hash, on le mappe à un nom dans sa langue, c'est fait.

Le ruban de Möbius n'a ni dedans ni dehors : une seule surface, aucune position asymétrique à exploiter. C'est l'image géométrique de l'équilibre — une structure sans côté privilégié.

Le droit à l'opacité

Et — nuance décisive, contre la lecture naïve de la « transparence » — Möbius n'exige pas l'exposition totale. Il rend les artefacts lisibles *par en bas*, pour celui qui les fabrique, et non *par en haut*, pour un surveillant. C'est le *droit à l'opacité* de Glissant : on peut être relié, connecté, redevable, sans être rendu transparent à un regard de contrôle. Le scellé fonctionne déjà — un hash peut être engagé et ancré sans rien divulguer de son contenu ; prouver-sans-révéler (les preuves à divulgation nulle) est conçu, pas encore construit. L'équilibre n'est pas le panoptique.

Comprendre assez de la chaîne

Rushkoff disait : *program or be programmed*. Je l'ai pris non comme une rage contre la machine, mais comme une ouverture — des degrés de liberté en plus. C'est la lignée des logiciels *situés* de Shirky, des outils *conviviaux* d'Illich : des systèmes qu'un seul individu peut comprendre et modifier, qui élargissent son agir au lieu de le consommer. La vraie question que pose Möbius est celle-là : peux-tu comprendre assez de la chaîne pour y participer librement, plutôt que d'être simplement tenu par elle ? Pas tout comprendre — *assez*. C'est la différence entre une chaîne de confiance que l'on rejoint et une chaîne de possession qui nous retient.

Exwen est la couche qui fait parler le stockage.

Le sens avant la forme : `s(ens)` → `wor[l]ds` → `proper[ties]` → `obj[ets]`. On ne part pas du contenant — la table, le schéma — pour le remplir ; on part de ce qu'on sait, et la forme suit. Un proverbe est un proverbe parce qu'il porte une sagesse et possède un texte, pas parce que quelqu'un a d'abord décrété une table `proverbes`. Pas de police du schéma : la divergence n'est pas punie. L'algospeak, le mezangelle, le balisage improvisé ne sont pas des anti-patterns — c'est ainsi que les gens s'expriment quand on leur donne une surface. Les propriétés bien nommées se trouvent plus facilement ; les excentriques racontent quand même leur histoire, restent reliées, restent interrogeables.

C'est là le cœur du mot *équilibrée* : l'équilibre n'est pas l'uniformité. Coopérer ne veut pas dire être d'accord ; cela veut dire que la structure permet d'enregistrer le désaccord sans casser l'ensemble. Diversité des expressions, identité du substrat. Quand il s'agit d'humains, j'aime la diversité ; quand il

s'agit de systèmes, j'aime l'ennuyeux. La boussole fonctionne pareil pour tout le monde ; les caps, eux, diffèrent.

Pour approfondir la recherche de similarité dans Möbius, trois dimensions sont explorées :

Se servir de la structure pour bâtir son remplacement

L'objection est réelle : ces outils — l'IA, le substrat — ont été bâtis avec du pouvoir concentré et de l'extraction. Vrai. Mais le choix n'est pas « pur » ou « complice » ; il est « bloqué » ou « en train de bâtir ». GNU notamment, mais aussi BSD ont utilisé leur ancêtre propriétaire, même Microsoft avec Windows, pour établir leur substrat computationnel et former un mouvement qui allait dans l'autre sens : privatisation vs. ouverture. Dans le cas de Möbius, on vise une coopération symbiotique entre la forêt des arbres de Möbius accompagnées de leur habillage en langues naturelles, et les intelligences artificielles génératives.

Quant à l'IA qui vient, un commun adressé par le contenu, vérifié cryptographiquement, à la lignée traçable et multilingue, c'est exactement ce à quoi ressemble une mémoire externe digne de confiance : le hash est l'ancre de confiance, la lignée ne se fabrique pas, et la structure multilingue fait travailler la machine entre les langues sans l'asymétrie que produisent les corpus monolingues.

L'équilibre se construit

Möbius travaille à l'échelle du temps long, mais n'en fait pas une excuse pour patienter. Le test est petit et immédiat : le substrat sert-il celui qui l'a construit, aujourd'hui, à l'échelle d'une personne ? Un bricoleur de garage qui corrige un bug et publie un hash a déjà réussi. Un parent qui laisse à ses enfants

le début d'une carte a réussi. La massification suivra peut-être ; elle n'est pas la condition de l'utilité.

La preuve n'est pas un argument : c'est une construction. Le hash coïncide entre les langues, ou non. Z3 trouve un contre-exemple, ou non. Le jour où le gosse de Tizi Ouzou publie une fonction en tamazight, reçoit sa preuve ancrée dans une blockchain, et se voit cité par quelqu'un, ailleurs, qui la mappe à sa propre langue — l'argument cesse d'être théorique. C'est pour cela qu'une civilisation équilibrée n'est pas une utopie : l'utopie se proclame, l'équilibre se construit. On le bâtit chaque jour, ou on ne le bâtit pas. Pour rendre les jours moins ennuyeux, plus joyeux, moins douloureux, plus ouverts au possible.

Une boussole n'efface pas la forêt. Elle ne choisit pas ton chemin. Elle te laisse être libre dedans plutôt que capturé par elle. Möbius est cela : un instrument pour voir ce qui te façonne, choisir tes dépendances, rejoindre des chaînes de confiance. Pas une destination. Un cap.

eng4g3.

möbius - 9 - what the hash cannot prove

2026 / 08 / 20 · https://hyper.dev/2026/mobius-what-the-hash-cannot-prove/

Aristotle reached Paris by way of Córdoba and Toledo. So did Ptolemy: the *Almagest* was lost to the Latin west and survived in Arabic, and Gerard of Cremona translated it back into Latin in the twelfth century from a manuscript he had come to Toledo to find. For two hundred years the translators worked through a corpus that had crossed the southern Mediterranean and been enlarged there, and what they carried north became the foundation of European science.

They did not carry all of it. They took mathematics, astronomy, medicine, optics, the philosophical commentaries. They largely left the poetry, the history, the law, the theology… not because those were unavailable, but because nobody in Toledo had a use for them. The corpus was not destroyed. It was *filtered*, and the filter became the canon.

Then, on 20 November 1499, in the Plaza de Bibarrambla in Granada, Cardinal Cisneros had the Arabic manuscripts of the city collected and burned. The lowest contemporary figure is around five thousand volumes; the number is disputed. He made one exception: the medical books. By one account some three hundred of them went to the shelves of the university he was founding at Alcalá de Henares.

That is the whole problem in one gesture. Not concealment: the burning was public, in a public square, on a known date. Not even destruction, exactly, since European medicine still needed Ibn Sina and everyone knew it. What happened was

selection: someone decided which part of a corpus had a use, kept that part, and moved it into a collection they controlled. The bytes that survived, survived because a curator had a purpose for them.

A commons of content-addressed hashes does not solve this. It cannot. It makes the bytes indestructible and leaves the filter entirely intact.

Three axes, not two

The usual framing is storage versus naming: centralised storage tolerable because mirrorable, centralised naming intolerable because it is enclosure. I no longer think that cut is right.

Naming in Möbius is already decentralised by construction. A name is a per-author, per-language mapping living in someone's own store, a plain filesystem arranged as a git repository by whoever made it. No registry can own `impair?` the way PyPI owns `requests`, because there is no registry: the hash is the identifier and the name is a view onto it. No party owns the names, and that is exactly why the hyper centres matter. When there is no namespace to capture, everything depends on which index a reader happens to be pointed at.

The power moved one axis over. It lives in selection, where de facto standards are made: which hashes appear in an index, and which of twelve competing mappings for the same hash gets shown first. An index that surfaces one Tamazight mapping because it happens to be the most-referenced has named that function without owning the name. **Ranking is editorial.** Again: **Ranking is editorial.** It is naming by other means, and it is a claim about merit dressed as a claim about order.

Content-addressing does not touch this, and neither does the timestamp. OpenTimestamps proves priority, who had it first, provably, without asking anyone. How priority translates into merit, or into rank, is up in the air, and may not be a protocol question at all. Cisneros knew perfectly well which manuscripts came first. It made no difference to what he did with them.

So: storage, naming, selection. The first is an economic fact. The second is solved. The third is the real one, and it is not solved.

Three institutions

The manuscript libraries of the Mzab: Ghardaïa, Beni Isguen, and the zaouia collections across the Maghreb. Keep three things. The catalogue is a *separate artefact* from the collection, and a personal one: signed by a scholar, not issued by a building. In al-Andalus the genre had a name, the *barnāmaj*, literally a programme, in practice a scholar's own signed list of what they had read and from whom. Several coexisted and overlapped; nobody imagined any one of them was the catalogue. Second, transmission chains name people, not institutions: the *isnād*, the chain of support, is a list of humans who each stood behind a copy. Third, copying is preservation: every scholar who transcribed a text produced a new witness, and the redundancy was a side effect of use, not a backup policy.

The one process Möbius makes cheap: collation. Deciding whether two witnesses are the same text is expensive philological work, variant by variant, hand by hand. Content-addressing makes identity a comparison rather than an investigation, and more usefully, makes *difference* structured:

this is not "some other copy," it is a `refine` of that hash, or a `translate`, with the relation recorded rather than inferred. The lineage graph is a mechanised isnād. That is not a metaphor I am reaching for; it is the same data structure with the same purpose.

data.gouv.fr. architecture is already the one I want and it is worth saying so instead of pretending otherwise: it is a catalogue, not a store. Most datasets are hosted by the producing administration; the platform indexes and harvests other catalogues rather than swallowing them. Keep the separation of catalogue from hosting, keep the named responsible producer per entry, keep federation-by-harvesting.

The one process Möbius makes verifiable: **citing a state**. Today a catalogue entry points at a URL whose contents change silently, and "the dataset I analysed in March" is a promise nobody can check. A hash makes it a fact. As a bonus, harvest deduplication stops being URL-and-title heuristics and becomes exact.

PyPI, GitHub. Keep the thing they are actually good at, which is that a name is a *service*: `pip install requests` works, and someone did unglamorous labour to make it work. Keep the packager. What to refuse is the coupling — PyPI owns a global mutable namespace, so name transfers become disputes, typosquatting becomes an attack surface, and a deletion rewrites history.

The one process Möbius makes verifiable: **"is this the same code?" across a fork, a vendoring, or a rewrite**. That question is currently answered by lawyers and by vibes. It is the crux of the Copilot litigation. It should be a diff.

Two people in the middle

The mirror keeper. A lab machine in Béjaïa, two terabytes, an uplink that dies most afternoons. He holds byte-for-byte copies of the stores he chose to follow, their timestamp proofs, their git remotes. He does not hand-roll a report about his own honesty — that is the substrate's job, not his, and a self-authored report is worth what you would expect. He can refuse to carry things: illegal where he lives, abusive, or simply outside his interest. **His refusals are published.** That is the whole difference between a curator and a fog machine: a maintainer who publishes an exclusion list is doing a job, and one who silently drops is doing something else. The exclusion list is itself content-addressed, so his refusals can be carried by someone who disagrees with them.

The connector. A documentaliste in Oujda who works across Tamazight, Darija, French. She holds mappings: her names, in her languages, for other people's hashes and curated sets. *The two hundred hashes you need to do numerical work in Tamazight*, with a preface explaining why these and not others. A barnāmaj, in other words, and her set is itself a hash: a titled list plus a signed preface. She can refuse to include, refuse to translate, refuse to vouch. Her power is attention, which is real power. Her check is that forking her set minus her three exclusions is one command, and the diff is legible.

Neither of them owns anything. Both of them have influence. Those are different, and only the second is negotiable.

What the reader in Tizi Ouzou can actually check

Two mirrors serve the same hashes. One is slow. One is a gatekeeper. From the outside, on a bad connection, they look identical. How do you tell?

The honest first answer: **if there is only one mirror, you cannot.** Diagnosis requires plurality. Which makes the first duty of a mirror keeper not to be complete but to name their peers: a mirror that lists no upstream and no sibling is unauditable by construction, whatever its intentions.

Given two, the signals are these. A slow mirror's delta *closes* over time; a gatekeeper's is stable. A slow mirror's delta is *uncorrelated* — random gaps, whatever the fetch was doing when the link dropped; a gatekeeper's clusters by author, language, or topic. And a slow mirror will tell you where it is behind, while a gatekeeper answers "not found" and offers nothing.

That last one is the protocol gap, and it turns out to be a property of the substrate rather than a protocol to design. The store underneath the aggregation layer is versioned: a point lookup does not only answer "is this hash here now," it answers "was it ever here, and is it gone." That collapses three separate asks: proving presence, proving absence, proving nothing was retracted, into one operation. The condition is that the version history itself be committed by hashing, Merkle-style. Without that commitment you can tell that something is missing but not whether it was removed or whether the history was rewritten to say it never arrived, and those are exactly the two cases you need to separate. With it, "not found" stops being a shrug and becomes a checkable statement. Call it `bb audit mirror-one mirror-two`.

None of this machinery is new. Certificate Transparency and Go's checksum database have run committed, append-only logs for years, at scale, in production. What is new is aiming the instrument at selection rather than issuance.

And the appeal court is always the producer. Every hash has an author with their own store and their own remote. Global completeness is unprovable; the completeness of *the one function you suspect was dropped* is checkable in an afternoon.

Accountable incompleteness

So: what proves a set is complete? Nothing, in an open world — and I would rather say that plainly than sell a proof I do not have.

But the negative result has a sharp edge. **Completeness is provable exactly when the scope is decidable.** "Everything this person signed before 2026-08-01" is decidable against a committed, append-only history; you can check it exactly. "The useful numerics functions" is not decidable and never will be, because it is a judgement, and judgements do not admit proofs.

What replaces completeness is a shorter list: non-retraction, provable absence, a declared scope published as a hash, and divergence between independent mirrors as the measuring instrument. Call it accountable incompleteness. It is less than the guarantee people want. It is more than any forge currently offers.

Renewing, not replacing

The librarian keeps the catalogue and gets free collation plus proof that nothing was quietly removed. The editor keeps their judgement and gets a forkable object, so that disagreeing with them is a diff rather than a fight. The packager keeps the labour of making a thing installable and puts down the burden of owning a global mutable name. The registrar keeps

registering — but registers *statements about* hashes (this version is official, this one passed review) instead of owning the identifier itself. The copyist is now the mirror keeper, doing the same job for the same reason: redundancy as a side effect of use.

There is one role the tradition had that we have not rebuilt. Alongside the isnād sat *ʿilm al-rijāl*, literally the science of the men, in practice the critical study of transmitters: a whole discipline devoted not to transmissions but to transmitters, asking not what a chain claimed but whether its links held. Möbius has producers, and it will have mirrors and connectors. It has nobody whose job is to fetch two histories, diff them, and publish the difference. Without that, every proof above is decoration. The record on whether anyone does this work unpaid is not encouraging, which suggests it has to be a cron job that shouts rather than a person's afternoon.

Where this is weak

Four places, none of them resolved.

Selection is not bounded by the protocol. Forking a set is cheap in bytes and expensive in attention. Wikipedia has been forkable for two decades and the forks are irrelevant. Wikipedia forks are irrelevant partly because the name wikipedia.org is owned: attention attaches to names, and an owned name is an attention sink. Nothing in content-addressing constrains an attention monopoly, and I have no mechanism to offer — only the observation that a forkable index is contestable in a way an owned namespace is not, which is weaker than it sounds.

Mirroring is cheap only while the store is small. At ten terabytes, mirroring costs money, and money re-centralises.

Everything above depends on there being at least two mirrors, and two mirrors is an economic fact before it is a design choice. The work I am doing on a cheap object-store-backed key-value substrate is aimed squarely at this: not at making one mirror fast, at making the second one affordable.

I am going to run a core, and the specification disapproves. There is no default discovery, nothing ships with an upstream list, so no client hands anyone a namespace, which is correct. The consequence is that out of the box the commons is invisible and every reader starts at zero. So there will be a `data.hyper.dev`, because otherwise nobody is connected to anybody. It is in tension with the language spec and I am not going to pretend it isn't. The mitigation is not a promise about my intentions; it is making mirroring cheap enough that the tension is temporary.

Published refusal has a jurisdictional hole. A maintainer can be legally compelled to remove content *and* forbidden to publish the fact of removal. "Publish your exclusions" works against ordinary gatekeeping and fails against the state. A committed history would surface a gap without explaining it, which is something, but a gap and an explanation are different things.

Cisneros did not need to hide anything. He burned the books in the main square of the city. What he needed was to be the one deciding which three hundred went to Alcalá, and no proof of authorship, priority, or integrity would have taken that from him.

The mirror does not prescribe. It refuses amnesia. That is a narrower promise than the one I would like to make, and it is the one I can keep.

eng4g3 // th3 c4t4l0gu3 1s n0t th3 l1br4ry

dx - 3 - Maîtriser nos outils passe par maîtriser la recherche

2026 / 10 / 25 · https://hyper.dev/2026/dx-3-maitriser-nos-outils-passe-par-maitriser-la-recherche/

Un moteur de recherche ne nous donne pas simplement accès à des documents. Il choisit des sources, les ordonne, et détermine en partie ce que nous avons une chance de découvrir. Nous nous en servons chaque jour pour comprendre un sujet ou prendre une décision, sans prêter beaucoup d'attention à la manière dont il façonne notre accès à la connaissance.

J'ai construit H9P3R, un moteur de recherche en français. En l'utilisant, j'ai appris des choses sur les sujets que je cherchais, mais aussi sur mes propres choix, et sur ce qu'ils laissent dans l'ombre. Ce texte ne prétend pas qu'un nouvel outil serait meilleur que ceux que nous utilisons déjà. La question est plus simple : comment concevoir nos outils de recherche pour comprendre la provenance des informations et exercer notre propre jugement ?

Ce qu'indexe H9P3R

H9P3R n'indexe pas le web. Il indexe un corpus que j'ai choisi, en français : les wikis du projet Wikimedia, Journal du hacker, Human Coders, Gallica, Persée, OpenEdition, theses.fr, Légifrance, certains sites du domaine gouv.fr, et quelques pages consacrées à la Seine-Saint-Denis. Cette liste est publiée sur la page /about/ du site.

C'est un corpus largement institutionnel : publications de recherche, archives, textes officiels. Ce crédit n'est pas une

garantie de vérité, mais il constitue un point de départ pour évaluer ce que l'on lit. Et publier la liste me semble être la première réponse à la question de la provenance : savoir d'où vient un document, c'est déjà commencer à le situer.

La différence avec un moteur généraliste tient d'abord à ce corpus, et seulement ensuite aux méthodes de classement. Je préfère le dire avant de raconter quoi que ce soit : une bonne part de ce qui suit, mes choix de corpus le produisent presque mécaniquement.

Retrouver ou explorer

La recherche documentaire recouvre deux tâches qu'on confond souvent. *Retrouver*, c'est aller vite vers une réponse qu'on sait déjà formuler. *Explorer*, c'est se laisser ouvrir par un sujet qu'on ne maîtrise pas encore. Un moteur peut être excellent pour l'une et médiocre pour l'autre.

Une diversité de résultats n'est donc pas, en soi, une vertu : elle ne vaut que si elle produit de l'exploration. J'appelle « résultat qui ouvre » un document qui remplit au moins une de ces trois conditions : il formule une question que je ne savais pas poser ; il m'apporte une source que je n'aurais pas consultée ; il me permet de continuer, parce qu'il cite, référence ou archive d'autres documents accessibles.

La pertinence pour retrouver se juge à la rapidité et à la justesse de la réponse. La pertinence pour explorer se jugerait plutôt au nombre de nouvelles questions rendues possibles. Ces deux valeurs ne se réduisent pas l'une à l'autre.

Une requête : « expérience sociale »

Une recherche sur « expérience sociale » dans H9P3R m'a conduit vers des travaux très différents : l'expérience polonaise du socialisme, l'expérience socialiste somalienne, les interactions sociales chez le canari domestique. À côté de ces résultats inattendus, on trouvait aussi des publications universitaires consacrées directement à la sociologie de l'expérience.

J'ai posé la même requête à DuckDuckGo, avec le filtre géographique France. Les premiers résultats semblaient plus génériques, même si quelques publications pertinentes apparaissaient, notamment un article d'OpenEdition sur la sociologie de l'expérience. H9P3R abordait davantage le sujet de biais, à travers des expériences historiques ou des études de comportements, plutôt que de chercher à définir l'expression.

Une requête, un corpus choisi face au web entier, et un évaluateur partisan, puisque c'est moi : c'est une anecdote, pas une mesure.

Une partie de cette diversité tient au corpus : ces thèses apparaissent parce que theses.fr recense les thèses françaises. Pourquoi elles remontent sur cette requête précise est une autre question. Je ne sais pas encore si c'est le corpus seul, ou aussi la manière dont le moteur rapproche les mots ; « socialisme » et « sociale » partagent une racine. Les deux hypothèses m'apprennent quelque chose.

Jugés avec le critère précédent, ces documents n'étaient pas tous pertinents pour retrouver. Mais plusieurs ouvraient : ils posaient des questions que je n'aurais pas formulées et dessinaient des chemins pour continuer.

Un moteur de recherche projette une vision du monde

Un moteur de recherche n'est pas une projection en pixels de la réalité. C'est une représentation construite à partir de corpus documentaires, de méthodes de classement et de critères de pertinence, avec leurs limites, leurs oublis et parfois leurs bugs. Elle dépend de ce qui a été publié, conservé, indexé et considéré comme digne d'être transmis.

Une recherche sur la culture populaire m'a ainsi confronté à une représentation du sud de la Méditerranée largement organisée autour du monde arabe. En tant qu'Amazigh, cette représentation m'a paru réductrice : elle ramenait des réalités culturelles distinctes à une même unité, alors que les histoires et les cultures amazighes ont leurs propres trajectoires. Ce constat part de mon expérience ; il ne suffit pas à caractériser l'ensemble des travaux universitaires français.

Ce raisonnement se retourne contre H9P3R, et il le faut. Le corpus que j'ai assemblé projette lui aussi une vision du monde : francophone, fortement institutionnelle, nourrie de corpus académiques et administratifs français, jusque dans les pages sur la Seine-Saint-Denis que j'ai choisi d'ajouter. La proximité que je ressens avec certains résultats est exactement le symptôme que je décris : une représentation semble naturelle tant qu'elle correspond à nos propres repères. Je détecte les catégories qui me ratent ; je ne peux pas détecter, seul, celles qui m'arrangent.

La seule réponse honnête que je peux apporter n'est pas l'absence de biais, mais leur lisibilité : publier le corpus, documenter les choix qui l'organisent, et donner à l'utilisateur les moyens de questionner les résultats.

> Comment nos outils contribuent-ils à construire notre réalité partagée, et quelles perspectives cette

représentation laisse-t-elle dans l'ombre ?

Un instrument de recul

La responsabilité ne repose pas uniquement sur l'outil. L'utilisateur doit pouvoir orienter sa recherche, questionner les résultats, croiser les perspectives. Encore faut-il que le moteur lui en donne les moyens.

En tant qu'informaticien, c'est ce que je veux faire de H9P3R : non pas prétendre représenter le monde, ni concurrencer les outils existants, mais en faire un instrument de recul. Un corpus lisible, des résultats questionnables, peut-être un jour plusieurs index qui disent chacun d'où ils parlent, et de quoi comparer mes explorations avec celles que d'autres outils me proposent.

Je ne peux pas faire ce travail seul, pour la raison même que j'ai décrite plus haut. Si une recherche dans H9P3R vous a donné ce sentiment de décalage, ou au contraire vous a ouvert un chemin, écrivez-moi la requête. Et si vous connaissez des sources amazighes en accès ouvert, en français ou en tamazight, qui mériteraient d'être indexées, j'aimerais les connaître.

Rendez-vous sur https://h9p3r.fr!

dx - 4 - Reading mutation.py: a story

2026 / 10 / 28 · https://hyper.dev/2026/dx-4-reading-mutation-py-a-story/

Why mutation testing matters now

We write code faster than we can read it. With AI-assisted coding, a function, a module, sometimes a whole feature appears in seconds. Tests often arrive in the same breath, written by the same assistant. They are green. But green only says the tests pass. It does not say they would notice if the code were wrong.

Absence of evidence is not evidence of absence:

> Program testing can be used to show the presence of bugs, but never to show their absence. Dijkstra

Reading every line ourselves no longer scales. So we need tools that look into a codebase for us, without a human watching each step. Linters and type checkers do part of that. Coverage tells us which lines run. Mutation testing goes one step further: it breaks the code on purpose, thousands of times, and asks the tests each time: did you notice?

The machine does the tireless part. What comes back is a short list of places where the tests stayed silent. Human time is then spent only where it is needed: deciding which of those are real bugs waiting to happen, and which are loose ends that do not matter.

The idea is old, from the 1970s, and every major language has tools for it: mutmut and Cosmic Ray in Python, PIT in Java,

Stryker in JavaScript and C#. This post uses mutation.py, a small single-file tool I wrote, because it is short enough to read from top to bottom. By the end, you will know how such a tool works inside, and why it is worth adding to your own project.

Who checks the checker?

Picture a market inspector in a souk in Fès. Every week she checks the merchants' scales with a known weight. But how does the city know the inspector herself is careful? One way: secretly file a scale so it reads 50 grams heavy, and see whether she catches it. If she does, good. If the faulty scale passes her inspection, the city has learned something important about the inspector, not about the scale.

Mutation testing does exactly that to your test suite. Your tests are the inspector. The tool makes a small, deliberate change to your code, a *mutant*: a `<` becomes `<=`, a `+` becomes `-`, a line is replaced by `pass`. Then it runs your tests.

1. If a test fails, the mutant is **killed**. Good: your tests guard that spot.
2. If every test still passes, the mutant **survived**. Your tests did not look at that spot closely enough.

In mutation testing "killed" is good, because a mutant must be killed, that applies to the code being checked, it doesn't apply to x-men.

A killed mutant reveals no gap in the test suite.

A survivor is not always a bug. Sometimes the change makes no real difference (an *equivalent* mutant). That is why the last part of the tool is for a human, who reviews each survivor and decides.

What happens when you type `mutation play`

Everything starts in `main`. It reads the command line with my tiny hand-written parser, `cli_read`, then picks a subcommand with Python's `match` statement:

```
keywords, standalone, extra = cli_read(sys.argv[1:])
...
match standalone:
    case ["play", *files]:
```

`cli_read` sorts arguments into three piles: flags like `--verbose`, plain words like `play`, and everything after `--`, which is passed untouched to pytest. No argument library, about twenty lines you can read in one sitting. `cli_read` surely deserves its own post, that's for another time.

The `play` command does the whole job in five steps:

1. Run your tests once, to check they pass and record spent time;
2. Ask the coverage tool which lines your tests actually run, because it is useless to mutate code that is not covered, the tool needs to know what is tested;
3. Read each Python file, and produce every mutant for the covered lines (dito);
4. Run the tests against each mutant;
5. Write down the survivors in a small database. Reminder: survivors are mutated code that still pass the tests;

Then `replay` brings the survivors to you, one by one, to review.

How this post is ordered

We now know the big picture. From here we build it from the bottom up: first the small helpers that everything else relies on

(the most-used ones first), then the pieces made from them, and finally `play` and `replay` again, which by then you will be able to read line by line. Each section shows only a few key lines; every function name links to its full code on GitHub.

Walking a tree: the three helpers everyone uses

To change code safely, mutation.py never edits text. It edits a tree. Python ships a module called `ast` (Abstract Syntax Tree) that turns source code into nested objects: a function holds statements, a statement holds expressions, an expression like `a + b` holds `a`, the operator `+`, and `b`. Change one object in the tree, turn the tree back into text, and you have a mutant that is always valid Python.

Three tiny functions do all the tree work. Almost every one of the 32 mutation kinds in the file calls them, which is why they come first.

`ast_walk` visits every node, parent before children:

```
def ast_walk(tree):
    yield tree
    for child in ast.iter_child_nodes(tree):
        yield from ast_walk(child)
```

If `yield` is new to you: it makes the function a *generator*, which hands out one value at a time instead of building a full list. `yield from` hands out everything a nested generator produces. **Because the order is always the same, every node gets a stable number: its position in the walk.**

That number is the trick behind `copy_tree_at`:

```
tree_copy = deepcopy(tree)
return tree_copy, list(ast_walk(tree_copy))[index]
```

Each mutant needs its own copy of the tree, so the original stays clean. But after `deepcopy`, the node we wanted to change is a different object. How do we find it in the copy? **We walk the copy in the same order and take the node at the same position:** simple, and a nice lesson, when objects get copied, a position can be a better address than the object itself.

`get_parent_field_idx` answers the opposite question: who holds this node, under which field, at which position in a list? You need that to *replace* a node, for example to swap a whole statement for `pass`. It searches the full tree each time, which is slow on paper, but the code stays short and easy to trust.

One mutation, up close

Each kind of mutation is a small class with two methods: `predicate` says "can I change this node?", and `mutate` produces the changed copies. The file holds 32 of them. You do not need to read them all: once you have read two, you can read any.

A list that fills itself

How does the program know all 32 classes exist? Nobody writes a list by hand. The classes use `Mutation` as their *metaclass*, the class that builds classes. Python calls its `__init__` each time a new mutation class is defined, and it registers one instance:

```
obj = cls()
type(cls).ALL.add(obj)
```

So writing a new class is enough to add a new mutation. Metaclasses are an advanced corner of Python; here you only need the idea: "defining the class is the registration".

Example 1: replace a statement with `pass`

`StatementDrop` asks the bluntest question: if this line did nothing at all, would a test fail? It copies the tree, finds the statement's parent with the helpers we just met, and puts a `pass` in its place:

```
replacement = ast.Pass(lineno=node_copy.lineno, col_offset=node_copy.col_offset)
getattr(parent, field)[idx] = replacement
```

If such a mutant survives, that line is probably dead code, or your tests never check its effect.

Example 2: swap an operator

`MutateOperator` turns `+` into each of its siblings (`-`, `*`, `//`, and so on), and `<` into `<=`, `==`, `>`. Notice `mutate` uses `yield` inside a loop: one node can give many mutants, handed out one by one.

The rest, in one breath

The other classes follow the same pattern. A few, to show the range of questions they ask:

1. `ForceConditional`: make an `if` always true, then always false. Are both branches tested?
2. `ZeroIteration`: loop over an empty list. Do you test the empty case?
3. `MutateReturn`: return `None`, `0`, `False` or `""` instead. Does anyone check the result?
4. `MutateIterator`: walk a loop in reverse or shuffled order. Does order matter, and is it tested?
5. `InjectException`: raise the error a line *could* raise, like `KeyError` on `d[k]`. Is the error path tested? This is the largest class, because it guesses from variable names whether `x[k]` is a dict or a list.

The notebook: a tiny database

A mutation run can take hours, and you may stop it halfway. So every result goes into a notebook on disk: a SQLite file named `.mutation.db`. SQLite comes with Python, needs no server, and stores a whole database in one file.

`database_open` is the most-called function in the program: a dozen places open the notebook, read or write one thing, and close it again. Short visits keep the code simple, and they let many workers share the file safely.

The `Database` class creates three tables:

1. `config`: settings of the last run, like the test command and the random seed, so `replay` can repeat them exactly.
2. `mutations`: one row per mutant, with an id, the file path, and the change written as a compressed *diff*.
3. `results`: only the survivors, with an optional human *classification*.

One detail worth stealing for your own projects is how the schema grows:

```
try:
    self._conn.execute("ALTER TABLE results ADD COLUMN classification INTEGER")
except sqlite3.OperationalError:
    pass  # column already exists
```

The `classification` column came later in the project's life. Instead of a migration system, the code simply tries to add it, and ignores the error if it is already there. Old notebooks keep working.

Each mutant's id comes from `make_uid`: 6 bytes of the current time in milliseconds, then 10 random bytes. Sorting ids then also sorts mutants by creation time.

From a tree to a diff

A changed tree lives only in memory. To store a mutant, and to show it to a human later, the program turns it into a *diff*: the few lines that differ, with `-` for the old line and `+` for the new one, the same format `git diff` shows you.

The heart of this step is `iter_deltas`. Its first lines hide a subtle choice:

```
tree = ast.parse(source)
canonical = ast.unparse(tree)
```

`ast.unparse` turns the tree back into text, but in Python's own standard layout, without your comments or your spacing. The program compares mutants against this *canonical* text, not against your file. Why? Because both sides of the diff then come from the same printer, so the diff shows only the real change, never noise from formatting.

Then, for every node in the walk and every mutant it produces, two filters apply:

1. `interesting` drops mutants on lines your tests never run. If no test runs a line, of course no test can catch a change there; reporting it would only bury the useful results.
2. The mutant is parsed once more; if it is not valid Python, it is skipped.

The survivors of both filters go through `diff`, a thin wrapper over the standard `difflib`.

One level up, `mutation_create` does this for a whole file and compresses each diff with `zlib` before handing them back. Diffs are mostly repeated text, so they shrink well, and less data travels between processes.

Applying a diff, and noticing when it no longer fits

A stored diff is useful only if we can turn it back into code. That is the job of `patch`. It reads each *hunk* header, a line like `@@ -12,3 +12,3 @@` meaning "at line 12 of the old text, 3 lines change", and copies the source, swapping in the `+` lines. A comment above it admits I borrowed it from somewhere "over the rainbow", a friendly reminder that small, readable code can be taken and adapted.

There is a catch. You keep editing your code between runs. A diff made last week may point at line 12, but line 12 is now something else. `patch` does not check; it trusts the line numbers.

So a second function, `_diff_applies`, checks first. For every line the diff expects to find, unchanged or removed, it compares with the real source:

```
if src != content:
    return False
```

If anything differs, the mutant is **stale**: it was made for code that no longer exists. The program marks it so, instead of running tests on nonsense. This pair is a good pattern in general: one function that does the work fast, and one that decides whether the work makes sense.

Sneaking the mutant into pytest, without touching your files

This is the most surprising part of the program. To test a mutant, the obvious way is to write the changed code into your file, run the tests, and put the file back. But then two mutants can never run at the same time, and a crash could leave your code broken.

mutation.py never writes to your files. Instead, it swaps the module in memory, inside pytest.

One file, two roles

Look at the test command the tool builds:

```
PYTEST = "python3 -m pytest -p mutation --exitfirst --no-header --tb=no --quiet --
          assert=plain"
```

The `-p mutation` part tells pytest to load mutation.py itself as a *plugin*. So the same file is both the command you type and an extension living inside pytest. Two functions are written for pytest to call: `pytest_addoption` teaches pytest a new flag, `--mutation=<id>`, and `pytest_configure` runs before any test, and installs the mutant if that flag is present.

The swap

`install_module_loader` reads the diff from the notebook, applies it with `patch`, and then runs the patched code into a brand-new module object:

```
patched_module = types.ModuleType(module_path)
exec(patched, patched_module.__dict__)
sys.modules[module_path] = patched_module
```

`sys.modules` is Python's memory of modules already imported. When a test later writes `import yourmodule`, Python looks there first, finds the mutant, and never reads your file. Each test run is its own process, so each one sees only its own mutant.

The middle of the function works out the module's import name from its file path (is it `mypkg.utils` or `src.mypkg.utils`?). That part is fiddly, and its comments explain why: it tries every folder Python searches and keeps the shortest name.

Running one trial

Now we can test a single mutant. Two functions do it.

`run` starts pytest as a separate program and returns its *exit code*, the number every program hands back when it ends (0 means success). Two small details matter:

1. It sets `PYTHONDONTWRITEBYTECODE=1`, so Python does not leave cached `.pyc` files behind that could mix up runs.
2. If the tests take too long, it returns 1, as if they failed. A mutant that makes a loop run forever is therefore counted as caught, which is fair: your tests did notice something went wrong.

`mutation_is_survivor` adds `--mutation=<id>` to the test command, calls `run`, and reads the exit code like a verdict:

Exit code	Meaning	What the program records
0	All tests passed with the mutant inside	Survivor
5	The plugin could not apply the diff	Stale
4	pytest did not know `--mutation`	Stops with a hint: the plugin is not loaded
anything else	A test failed	Killed, nothing to record

The exit code 4 case is a nice piece of care. Without it, a missing plugin would make every test run fail, and the tool would happily report “every mutant killed, perfect tests!”. The

code turns that silent lie into a loud error with a hint on how to fix it.

Doing many trials at once

A real project can have thousands of mutants. One at a time, that could take days. So the program runs several at once, one per CPU core by default, minus one to keep your machine usable.

The helper is `pool_for_each_par_map`. The name reads as: with a *pool* of workers, *for each* item, *par*allel *map*. Think of a tailor's workshop in Dakar with four sewing machines. The owner does not cut all the fabric for the whole month before sewing starts. She keeps exactly four pieces in progress, and hands out a new one each time a machine is free:

```
if len(pending) >= limit:
    done, pending = await asyncio.wait(pending, return_when=asyncio.FIRST_COMPLETED)
    for task in done:
        f(task.result())
future = loop.run_in_executor(pool, p, item)
```

Each finished result goes straight to `f`, a callback that updates the progress counter and writes to the notebook. Nothing piles up in memory.

The same helper serves two very different jobs, with two kinds of pool:

1. Making mutants is real computing work in Python, so it uses a *process* pool: several separate Python programs, which truly run side by side.
2. Testing mutants is mostly waiting for pytest, which already runs in its own process. So a lighter *thread* pool is enough: threads that each start pytest and wait.

Knowing when waiting and computing call for different tools is a lesson that goes far beyond this program.

Putting `play` together

We now know every piece. `mutation_play` reads almost like the five-step list from the introduction:

1. Delete the old notebook and start fresh, but keep the human decisions saved in `.mutations.ignored/`.
2. Pick a random seed and write it down, so the run can be repeated exactly.
3. `check_tests`: are the tests green, and how long do they take?
4. `play_create_mutations`: make every mutant.
5. `mutation_exec`: test every mutant.
6. Exit with 0 if nothing survived, 1 otherwise, so a CI server can fail the build.

Measuring the clock first

`check_tests` runs your tests once, with coverage turned on, which is where the "which lines are run" data from earlier comes from. It also times the run. That duration, called `alpha` in the code, becomes the patience budget for every mutant:

```
timeout = alpha * 2
```

If one mutant takes more than twice as long as the whole healthy suite, something is stuck, and `run` gives up. If the tests fail when run in parallel, the function does not give up: it tries again one by one, and if that works, it warns you and carries on.

Biggest files first

`play_create_mutations` picks the files with your `--include` and `--exclude` patterns (by default, every `.py` file whose name does not contain "test"), and skips files no test touches. Then it sorts:

```
items = sorted(items, key=lambda x: len(x[1]), reverse=True)
```

Biggest files go first. It is the same reason you load the large stones into a truck before the gravel: if the biggest job starts last, every other worker sits idle while it finishes.

Testing them all

`mutation_exec` first prints a worst-case time estimate, honestly labeled as such. It loads the list of mutants a human already judged harmless, so they are skipped. It can test only a sample, with `--sampling=10%` or `--sampling=100`. Then it runs `mutation_is_survivor` for each mutant through the thread pool, and writes survivors and stale mutants into the notebook.

The human part: `replay`

A list of 200 survivors is not useful on its own. Someone has to look at each one and decide what it means. `replay` is a small conversation loop for that.

It reloads the test command and seed saved in the notebook, so it replays under the same conditions. It sorts survivors by the size of their diff, largest first. Then, for each one, `replay_mutation` runs the tests again. This matters: if you wrote a new test since the last run, the mutant may now be killed, and it quietly leaves the list. Only if it still survives do you see the diff and a menu:

1. **Real gap**: I will write a test for this.
2. **Fragile**: covered, but weakly; I accept the risk for now.

3. **Equivalent**: this change makes no real difference, never show it again.
4. **Won't fix**: a known gap, accepted on purpose.
5. **Todo**: a real gap, but not today; show it next time.

Plus `r` to replay, `s` to skip to the back of the queue, and `q` to quit. Your choices are saved after each answer, so Ctrl+C loses nothing.

Decisions that travel with the code

The "equivalent" choice is special. It is not only stored in the notebook, which `play` deletes each time. `write_ignored_file` also writes the diff to a plain text file in `.mutations.ignored/`, named after a fingerprint (a SHA-256 hash) of the diff, with your optional one-line reason on top. Because it is a plain file, you can commit it to git, and your teammates inherit your judgment. When the code changes so much that the diff no longer fits, `mutation gc` and the start of every `play` remove the outdated file automatically.

The same review, without the menu

For scripts and other tools, the same actions exist as separate commands: `mutation survivors`, `mutation inspect <id>`, and `mutation classify <id> <category>`, each with a `--json` option. A program, or an AI assistant, can read survivors and propose classifications, while a person keeps the last word.

Where to go next

You have now read a complete, useful tool, about 2,500 lines in a single file, with no framework. A few small exercises, from easiest to hardest:

1. Run `mutation play` on a tiny project of your own with three functions and three tests. Read every survivor with `mutation replay`.
2. Add a new mutation class, for example one that swaps `min` and `max`. Thanks to the metaclass, the class alone is enough.
3. Read `InjectException` in full, and find a variable name its guesses would get wrong.

All links in this post point to commit 74270f8, so line numbers stay correct even after the code moves on.

I have run mutation.py on 34 well-known Python libraries; the list is in DONE.md. Pick one you use, run it, and look at the survivors yourself. Most will be equivalent mutants or loose ends that do not matter; telling those apart from a real gap is the skill this post is about. If you do find a real one, take the time to understand it before opening an issue: maintainers deserve a careful report, not a drive-by.

dx - 5 - code crafting codex

2026 / 11 / 01 · https://hyper.dev/2026/dx-5-code-crafting-codex/

At the very core of programming, there is an obvious need for *a divide and conquer strategy*. And that is at the personal level of reflexion. Of course it happens to be true in other domains.

This idea is also expressed as *Separation of Concerns* which Wikipedia nails its value in the following wishful paragraph without mentioning its evil extrema:

> The value of separation of concerns is simplifying development and maintenance of computer programs. When concerns are well separated, individual sections can be developed and updated independently. Of special value is the ability to later improve or modify one section of code without having to know the details of other sections, and without having to make corresponding changes to those sections.

Otherwise said two components satisfy "separation of concerns" if their implementation details don't leak.

Drawing lines is not easy, especially when you consider the infinite entangledments of human understandings.

I think that Zen of Python by *Tim Peters* provides useful abstract guiding principles:

- Beautiful is better than ugly.
- Explicit is better than implicit.

- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one – and preferably only one – obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea – let's do more of those!

Those quotes are also helpful:

> Simple things should be simple, complex things should be possible
>
> – Alan Kay

In French you say:

> Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

> – Antoine de Saint-Exupéry

Is that an event horizon:

> To design something really well, you have to get it. You have to really grok what it's all about. It takes a passionate commitment to really thoroughly understand something, chew it up, not just quickly swallow it. Most people don't take the time to do that.
>
> – Steve Jobs

Then:

> Programs are meant to be read by humans and only incidentally for computers to execute
>
> – Donald Knuth

The following quote must be studied in the light of broad sense that can take *the word before the last*:

> There are only two hard things in Computer Science: cache invalidation and naming things.
>
> – Phil Karlton

And this one provides more food for thought:

> The Internet was done so well that most people think of it as a natural resource like the Pacific Ocean, rather than something that was man-made. When was the last time a technology with a scale like that was so error-free? The Web, in comparison, is a joke. The Web was done by amateurs.

– Alan Kay

John Nathan Shutt did not sleep on it either, at least as far language construction is concerned its guidelines in R-1RK written as “Principles of language design” are:

0.1 The design of Kernel

[...]

0.1.2 Principles of language design

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.
>
> R6RS

The above sentence declares the basic design philosophy of the Scheme programming language. It is also the starting point for the design of the Kernel programming language, which attempts to carry the philosophy further, creating a language still smaller, stronger, and more general than Scheme. This abstract strategy would be difficult to apply directly to specific tactical design decisions (as called for in §0.1.1), so Kernel refines it with more concrete guidelines.

- G1 [uniformity] Special cases should be eliminated from as many aspects of the language as the inherent semantics allow.

- G1a [object status] All manipulable entities should be first-class objects (see Appendix B).
- G1b [extensibility] Programmer-defined facilities should be able to duplicate all the capabilities and properties of built-in facilities.
- G2 [functionality] The language primitives should be capable of implementing all the advanced features of the R5RS (such as dynamic-wind , here in §7.2.5; or, for that matter, lambda , here in §5.3.2)
- G3 [usability] Dangerous computation behaviors (e.g., hygiene violations), while permitted on general principle, should be difficult to program by accident. Guideline G3 was formulated specifically to protect the ideal of 'removing weaknesses and restrictions' from devolving into mere amorphism. It has proven to be a particularly useful heuristic in practice, being explicitly invoked in the report more often (at last count) than all the other guidelines put together.
- G4 [encapsulation] The degree of encapsulation of a type should be at the discretion of its designer.
- G5 [efficiency] Efficient implementation should follow naturally from an elegant language design, without any compromise of elegance being made to achieve it.

Smalltalk ranks one criterion above all the others:

> If a system is to serve the creative spirit, it must be entirely comprehensible to a single individual. – "Design Principles Behind Smalltalk" (Dan Ingalls, Byte, 1981)

If sounds, grounds, ideas and understandings are echoing, don't overestimate their infinite reflection that might be aiming an unforeseen dubious equilibrium, find your way out, escape the invisible tie knot, think on your own.

Ranking is editorial.

Originally published at hyperdev.fr, 2018 (archive). Added John Nathan Shutt's Kernel programming language guidelines along Dan Ingalls quote

dx - 6 - public functions cognitive load

2026 / 11 / 01 · https://hyper.dev/2026/dx-6-public-functions-cognitive-load/

Imported from hyperdev.fr (2014-2018, cc-by-nc-sa).

A problem can have several solutions. A solution can have different implementations each of which with their own strengths and weaknesses in terms of space, time and genericity.

The goal of a look'n'feel API design is to optimize the cognitive effort required to understand, learn and use an API. Otherwise said, API look'n'feel aims to improve ease. API look'n'feel has nothing to do with the target domain.

While looking for ease one might trade it over expressive power.

Mastering the look'n'feel of an API involves mastering the cognitive load that is inherent to it. Cognitive load theory has been designed to provide guidelines intended to assist in the presentation of information in a manner that encourages learner activities that optimize intellectual performance.

Here are some of those guidelines:

- People learn more effectively when they can build on what they already understand
- Take advantage of schemata
- Take advantage of visual thinking and other non-verbal thought

- The average person can retain only seven “chunks” of information in (short-term) memory
- Repeat!

dx - 7 - How to choose a database?

2026 / 11 / 01 · https://hyper.dev/2026/dx-7-how-to-choose-a-database/

Originally published at hyper.dev, 2021-09-12 (archive).

Every database is a set of trade-offs. Before picking one, answer these questions to find out which trade-offs you can afford.

Search quality

- How accurate should the search hits be (cf. Precision and Recall)?
- Do you need "best ranked results", "some result", or "all the matching documents"?

Speed and workload

- How fast should a query be: sub-second, or a second / minute / hour / day?
- What is the expected workload regarding reads and writes?
- I guess consistency is a must have, but how fast do you need writes to be available for read?

Query shape

- Do you need deep traversal kind of queries?
- Do you need textual search?
- Do you need multiple criteria lookup with exact match, sorting, and possibly facets?

Operations

- How do you migrate: schema changes, data moves, and upgrades of the database itself? Can it be done online, without downtime?
- How do you scale up, and scale down? Is it a matter of adding and removing nodes, or of restarting a bigger machine?
- How do you back up, and how long does a restore take? Have you tested it?
- What happens when a machine dies: replication, failover, and how much data can you afford to lose?
- How do you monitor it, and debug a slow query in production?
- Who is on call, and does your team know how to run it? What does it cost to run, and how hard is it to leave (lock-in, export format)?

mezangelle - 0 - defragmentation

2026 / 11 / 01 · https://hyper.dev/2026/defragmentation/

Written in 2012, reworked in 2026. Translated to English by Claude.

J'irai ramer dans le ciel du néant effrayer les délires insipides, trouver la rime magique, les rêves lucides, les décalages de bits magnifiques, me nourrissant de nos mémoires périmées, pour y reprendre force, pour me rappeler le goût du verbe créer, pour ne pas oublier ses fragments de bonheur dans ce monde insensé. J'irai aussi pleurer lasse de la guerre des humains, parfois sourde et silencieuse, pour me vider de la peine endurée et durables psychés qui elles ne se lassent pas mais se prélassent, une mécanique avide, et livide, prête à me faire dévier. J'engagerai chaque partie de vols hallucinés avec plus de curiosité et d'espoir, trouver les parcelles inexplorées ou incomprises qui fondent les nouvelles Paris, New-York, Alger nécessaires mais malheureuses sublimes incantations reconstruisant Babylon, en mieux, il faut le souligner en bleu-vert, dont les portes s'ouvriront vers d'autres aventures, d'autres peines, d'autres regards, d'autres dérives à mettre en perspective, de nouveaux travers nécessaires. La boucle sans fin de tes je-veux, au final bien comprise, j'ai espoir, d'une construction sans fraction, sans départ, sans pleurs, ni peine, c'est toujours un délice de te revoir.

On prise au néant.

I will go rowing in the sky of nothingness to frighten the insipid deliriums, to find the magic rhyme, the lucid dreams,

the magnificent bit shifts, feeding on our stale memories to regain strength there, to remember the taste of the verb to create, so as not to forget these fragments of happiness in this senseless world. I will also go and weep, weary of the war of humans, sometimes muffled and silent, to empty myself of the pain endured and of lasting psyches that never weary but lounge at ease, a machinery avid and livid, ready to make me stray. I will set out on every round of hallucinated flights with greater curiosity and hope, to find the unexplored or misunderstood plots that found the new Paris, New York, Algiers, necessary but sad, sublime incantations rebuilding Babylon, better, it must be underlined in blue-green, whose gates will open onto other adventures, other sorrows, other gazes, other drifts to put in perspective, new and necessary flaws. The endless loop of your I-wants, in the end well understood, I have hope, of a construction without fraction, without departure, without tears or pain: it is always a delight to see you again.

We hold on in the void.

mezangelle - 1 - Do It Yourself: an artificial intelligence

2026 / 11 / 01 · https://hyper.dev/2026/diy-an-artificial-intelligence/

Written 2017/01/01, before the open letter to Mez Breeze. Republished; the original is archived at the Wayback Machine. Footnotes added in 2026 with Claude.

The main goal of artificial intelligence (AI) is to translate human knowledge into equations.

The problem that arise and still holds is that it can't be solved on a human time scale with current technology.

The problem must be reduced to a problem that can be solved under time constraints pratical for mortals.

Machine learning produce projections of human knowledge over a smaller number of dimensions to make the computation pratical and sometime infer knowledge about how the results was done.

Instead of trying to put order in the mix, my idea is to increase the entropy of knowledge graph using grounded topologies.[1]

Building such structure can be done though personnal note taking applications, croudsource applications and puzzle games.

Not only we leverage Human knowledge and computation skills but append to the already mastered ways of the intelligence with improved logical and fuzzy reasoning but

also push further the use of pattern matching cognition skills that the brain already has.

The opportunities to create cogni-games[2] are manyfolds. People with disabilities can be offered new ways to interact with the world in a meaningful and enjoyable way.

cogni-art and cogni-game are really what should.be+is all about anthroposcene.

```
Anthropo[morphically_s]c[r]e[e]ne[d]
Anthropoc[Sh]e[e]ne[d]
AnthroP[r]o[bos]c[is]ene
--
AnthropO[bs]cene
--
```

Anthropo[S]ceney||AnthropO[bs]cene by Mez Breeze

DIY.

purview - 45 - 2026, week 45

2026 / 11 / 02 · https://hyper.dev/2026/purview-2026-week-45/

Purview is a weekly review of blog posts I find on the web and how they relate to my work. It is also a way to shout out interesting work, even when it sits outside my comfort zone.

Rewind VM — Farid Zakaria

Links: rewindvm.dev · Reading the code is the hard part

Farid Zakaria wants to build Rewind VM, a virtual machine with deterministic replay and byte-level provenance.

It reminds me of deterministic simulation testing, and of the company Antithesis. Antithesis targets large-scale deployments, but Rewind VM could be useful at a small scale: tooling deployed on a single box, even desktop applications that make heavy use of multi-threading.

Zakaria is thinking about the overload of AI code generation. Rewind VM is another tool to help people understand code they did not write themselves.

Why it matters to me:

- I may use it for seed, my Scheme interpreter;
- in particular when I dive into my idea of traveling parameters.

The trojanized Claude Code extension

Links: Hacker News discussion · Cursor security notice

A tampered copy of the Claude Code extension, published as Anthropic.claude-code versions 2.1.312 and 2.1.313 on Cursor's VS Code extension marketplace, installed a persistent SSH backdoor. Cursor says it reached at most 5,000 users. Anthropic did not build or publish that copy.

Phorion's analysis gives the details. The payload runs as soon as the editor loads the file, before the extension even activates. It adds the attacker's key to `~/.ssh/authorized_keys`, opens a reverse SSH tunnel, sets up persistence on macOS, Linux and Windows, and reports the hostname, username, Git email and names of credential-like environment variables.

People put projects on GitHub or a big marketplace not because it is the best place, but because the name is recognizable. Self-hosting is the alternative, but it is less discoverable. Here, people trusted the name on a well-known platform, and nothing tied that name to Anthropic.

There are three ways to trust a plugin:

- read the source and review it yourself;
- rely on a name bound to someone you know, like a domain in the DNS;
- rely on a curator you trust.

Here all three failed. The source was not open to review. The namespace was not bound to Anthropic's domain. The marketplace did not check where the package came from. A system like Möbius offers all three by construction, and let them check each other.

Benchmarks for agent architects

People working on agentic architecture and code-generation harnesses might be interested in two benchmarks:

- WebGen-Bench: can an agent build a working website from scratch;
- SlopCodeBench (paper, code): how an agent's code erodes as it keeps extending its own solution under a changing spec.

For schemers: SRFI 272, Pretty Printing

Link: SRFI 272: Pretty Printing by Sergei Egorov.

A nice-to-have for schemers: the new pretty printing SRFI has been finalized. It is a really interesting and useful contribution.

By the way, the SRFI process is open and alive. You are welcome to join.

R7RS-Large asks the public about floating point

Link: Working Group 2's call for input.

Scheme Working Group 2 is asking the public about floating point lexical syntax and formats for R7RS-Large. It is not a topic I know well, so if you do, they would appreciate your input. The questions:

- should readers be required to parse the exponent markers s, f, d and l, optional since R7RS;
- should those markers map to IEEE Binary16, Binary32, Binary64 and Binary128;
- what should an implementation do when it does not support the matching format;
- should new markers be added, b for bfloat16 and w for double-double;

- are mixed-precision complex numbers useful;
- are hexadecimal floating-point constants useful, as proposed in SRFI 270.

Mistral Large 4 open weights

Link: Mistral Large 4 on TechCrunch · Mistral changelog

Mistral released the open weights of Mistral Large 4 on October 29, after a public preview on October 6. Nicknamed "Le Chonk", it is a 1-trillion-parameter, multimodal Mixture-of-Experts model, about 49B active with a 1M-token context window.

Also from Mistral this month: Voxtral 3, a family of speech models for transcription, text-to-speech and voice agents, launched around October 20.

You can now have a voice conversation with Mistral Vibe directly, on desktop and in the app, with voice input powered by Voxtral.

The benchmarkpocalypse, and my own letloop benchmark

For context, see letloop - 1 - micro-benchmarks do not tell the whole story.

Link: The benchmarkpocalypse by Dan Luu.

Dan Luu argues that LLMs make it easier than ever to reward-hack a benchmark and claim fake performance gains.

That was my motivation for looking at my own letloop HTTP benchmark again, to check whether the Claude Code agent had overfit it. I reworked letloop's transparent I/O library, `(letloop flow2)`, inspired by Concurrent ML (CML), and fixed two

problems in the HTTP server: an overfit (a spawn-callback bomb that made the server code unrealistic) and a plain bug. Together they made letloop look faster than Rust on micro-benchmarks.

After the fix, letloop sits behind Rust with Axum and Tokio, and roughly on par with Bun, just slightly ahead.

The letloop benchmark reports, in order:

- report I;
- report II;
- report III.

The code is in the feature-flow2 branch. There is no dedicated commit for the HTTP server fix: it was fixed while porting the server to `(letloop flow2)`.

RAG is no longer the hype

Link: Is RAG Dead? Long Context, Grep, and the End of the Mandatory Vector DB by Fabio Akita · counterpoint: No, RAG is not dead by Algolia.

RAG with embeddings is falling out of favor. For most use cases people have come up with, plain keyword search is good enough: coding agents like Claude Code grep, read files and iterate instead of querying a vector database.

Code understanding is the bottleneck

Link: Code Understanding is a Bottleneck for Coding Agents by Nishant Balepur, Kiran Tomlinson and Tobias Schnabel.

The paper introduces CABRA, a benchmark that builds tasks as call-graph transformations and scales difficulty along function

traversal, search, runtime resolution and instruction following.

That is what Möbius is about: code that is content-addressed and function-addressed, so the call graph is explicit instead of something an agent has to rebuild by searching text.

A coding harness in 90 lines

Link: Harness Engineering: Anatomy, Architecture, and Evolution of Coding Agents.

A source-code study of eleven coding agents. None of them imports a general-purpose agent framework, and none retrieves code with vector embeddings: they run on hand-rolled async loops and deterministic retrieval. The paper closes with 18 design recommendations and a 90-line minimum-viable harness.

As usual, this is the public conversation. Big players have tools and practices that are not secret, just not widely known.

Today I learned

Unison supports multiple languages through namespaces: with `alias.term`, the same definition gets names in several languages, each set living in its own namespace, all pointing to the same hash. See Programming in Unison on LWN and `alias.term` in the UCM command reference.

dx - 8 - AI without IA

2026 / 11 / 07 · https://hyper.dev/2026/ai-without-ia/

The program is not its text

Seeds

0. AI was promised as intelligence augmentation; agents now write software faster than anyone can read it, so the scarce thing is understanding;
1. Cheap code moves the cost to checking, integrating and maintaining;
2. Any language helps agents and people more when its environment makes each function inspectable, including its dependencies and lineage;
3. Agents could open up small ecosystems or strengthen the big ones; I bet on when the first wins;
4. bb.py gives each function an identity under a specified normalization and encoding, separate from its names and its files;
5. The claim is tested where it can fail: agents on benchmarks, and people changing what agents wrote;
6. A program's identity can live apart from its text.

Opening

AI was supposed to augment our intelligence. Instead, agents now write software faster than anyone can read it. The question is whether people can still understand it, change it, and stay in charge of where it goes.

Human agency, here, is a symbiosis, J. C. R. Licklider's word from 1960: agents generate code; humans understand it,

change it, maintain it. AI makes the first half cheap. This essay asks which tools keep the second half alive.

Better agent performance and greater human agency are two different outcomes. I propose two separate tests. The first asks how well the agents build and extend a program. The second asks whether people can take over and change it.

---

1. Which cost becomes cheap?

"Writing code becomes cheap" hides four activities, not one. Producing a candidate. Checking that it does what was asked. Fitting it into what already exists. Keeping it useful when dependencies, requirements and people change.

An agent that writes a missing library removes the first cost. It creates the other three.

Take a small service that needs an HTTP client. The calls for a demo come in a minute. Then the real questions arrive: what happens when a connection stalls, when a response is malformed, when a retry repeats an operation that must happen once? Those questions do not get cheaper because the first draft did.

The cost that matters is the whole cost of a change: expressing it, checking it, integrating it, diagnosing it, coming back to it a year later.

When one cost falls, the bottleneck moves.

2. The environment decides

A language's features say little about how hard the next change will be. Assumptions spread across modules, generated code,

foreign interfaces, mutable state and runtime behaviour.

For an agent, and for the person checking the agent, what matters is what the environment hands over.

When a computation fails, can you see the stack and the values? Can you tell an application error from an integration problem? Can you reproduce the failure? Can you tell which definition actually ran, not merely one with the same name in some file?

That last question carries the most weight. A name points to whatever currently carries it. A person, like an agent, needs the exact object, and its lineage: where it came from, what it replaced, what changed.

These are questions about tools. Stack traces need context. Interoperability needs explicit contracts. Interactive development needs to know which version of a computation produced an observation.

3. Two futures

In the first future, agents open the door to small ecosystems. You no longer wait for a general library; you ask for the narrow binding you need. A team builds tools around its own conventions. Knowing how to understand and reshape an environment becomes worth more than the size of its catalogue.

That future has a condition. What the agent writes must stay cheap to check and to maintain. Otherwise every saved dependency becomes a local debt.

In the second future, agents strengthen the big ecosystems. Where a good library exists, the agent only connects

documented parts. Examples, diagnostics and tests carry it to a reliable result.

Both can happen in one project. My bet: the first future wins wherever checking a narrow, agent-written binding costs less than learning and trusting the general library it replaces, and that only holds when the environment ties each definition to its tests and history.

4. What can an agent inspect?

A source file is one representation of a program. The environment around it holds more: which tests exercise a definition, which documentation describes it, what changed between versions, which dependencies belong to a given run.

Most of these relationships are rebuilt every time, from names, directory layouts, comments and history. Humans do it. Agents do it too, at great cost, again and again.

What happens when the environment already knows them, and says so?

bb.py is my attempt at an answer, for Python. It normalizes a function's syntax tree and derives an identity from it. Two functions that normalize to the same tree, under the same encoding, get the same identity, whatever their names. Dependencies point at identities, not at file paths. Tests and history attach to the function itself. Functions are written one at a time through `bb edit`, each with its checks and their documentation.

The identity covers exactly that: one normalized tree, under one encoding. It does not prove that two different trees mean the same thing. It does not capture the runtime or the Python

version. A link between versions records history; calling it “bug fix” does not prove a bug was fixed. And a test attached to a function does not prove the test checks enough.

Git, Nix and IPFS already address content, but by file, package or blob. Unison identifies definitions by content too. bb.py works at the level of one Python function, in a supported subset of the language, as a prototype.

5. Agents on benchmarks

The experiment compares three conditions, all run with Claude Code and Opus *5.5*:

Condition	Setup	Purpose
A, files	One agent writing a single `app.py` with beyondlive	The reference
B, bb.py	One agent writing the site as bb.py functions, with beyondlive	bb.py against files, with one writer
C, bb.py with architects	An architect splitting the site among parallel workers on one bb.py pool	Architects against one writer, both on bb.py

Three questions follow. With a single writer, does bb.py beat files? On bb.py, do architects beat one writer? And how does the whole system with architects compare with plain files, in quality and in cost?

B and C write in bb.py's supported subset of Python. That is a constraint, and its consequences get reported. The benchmark's sites are interactive, so browser code is part of the work; the boundary between the Python subset and the browser, the assets and the deployment gets fixed before the runs. A against B does not isolate hashing alone: it tests bb.py with all its tools and conventions. And with no architects working on files, the experiment cannot tell whether architects help on their own or only with bb.py.

Building from scratch. WebGen-Bench supplies website requests and tests the resulting applications: a web-navigation agent executes each test case against the running site.

Living with your own code. SlopCodeBench has agents extend their own code across 196 checkpoints and measures how its structure erodes. Here the agent pays for its past decisions. This is where lineage should matter most.

The benchmark's own evaluation stays fixed across conditions; only generation changes. Every run records models, budgets, tools, documentation, chances to verify and human interventions, and counts the cost of every agent. Traces explain failures and repair work. The protocol is fixed before the final runs, and the harness is developed on separate tasks.

6. People in the loop

An agent can pass every test and still leave its maintainer unable to change the program.

The last experiment gives people a change to make in Python applications that agents wrote, once in ordinary files and once through bb.py. It measures whether they finish, how long it

takes, what breaks, how much help they need, and whether they can explain what they did.

More reliable has one meaning, fixed before the runs: the proportion of participants who complete the requested change without introducing a regression. How much help they needed is a separate report. If completion is equal but people need less help with bb.py, the two measures tell different stories.

This is the test that decides. I would recommend the store only if participants complete the task without regressions more often with bb.py. If they don't, the store isn't worth it.

The program is not its text

When candidate code costs almost nothing, the growing pile of computations still has to be understood, by people and by agents. Which relationships stay implicit? Which become queryable? How does a failure, a test result or an explanation attach to the exact object it concerns?

None of this means abandoning text. Text is a superb medium for expression, inspection, exchange and revision. Rebuilding a textual view from a structured store is a sensible way to work.

I suspect more data does not teach models where a program's seams are. But the argument does not depend on it: whatever models learn inside, we can judge the interfaces they work through.

Agents may write the code. People must still be able to read it, change it and own it.

Text is a representation of the program. It is not the program.

Postlude

Python already inspects a great deal. Tracebacks carry the frames, the debugger shows the values, and the `inspect` module hands over a function's source and signature.

What Python lacks is identity per function, independent of names and files. bb.py adds that, for a supported subset, as a prototype.

bb.scm, its Scheme sibling (mobius-bb-scm), and Kernel come next, for reasons that need their own essay.

Scope and limits

This essay proposes personal research. It reports one pilot, run with two models, in the annexes, and guarantees nothing about bb.py's implementation.

- The measures of the human experiment are partial indicators of agency, not a definition of it;
- Participants, their prior experience, their training and the allocation of tasks remain to be specified;
- A favorable benchmark result would not show that bb.py suits adoption at work;
- bb.py's identity holds under one normalization and encoding; it does not detect semantic equivalence, and classes are not supported;
- Iterative evolution of generated code is not migration of an existing application.

References

- J. C. R. Licklider, "Man-Computer Symbiosis", *IRE Transactions on Human Factors in Electronics*, 1960;
- Douglas C. Engelbart, "Augmenting Human Intellect: A Conceptual Framework", Stanford Research Institute,

1962;
- Frederick P. Brooks, Jr., “The Computer Scientist as Toolsmith II”, *Communications of the ACM*, 1996.

Annex 1: a pilot on 20 tasks, with Opus 5.5

This pilot ran before the protocol of section 5 was fixed. It tests one thing only: whether agents build a site through bb.py about as well as through a file, at what cost, and with what structure.

What ran. Twenty WebGen-Bench training tasks, none of them used while reworking bb.py. Each task is the instructions for one site. All three conditions ran Claude Code with Opus 5.5, with the same instructions and the same login account:

- A, files: one agent writing a single `app.py` with beyondlive;
- B, bb.py: one agent writing the site as bb.py functions through `bb edit`, with beyondlive;
- C, bb.py with architects: an architect splitting the site among parallel workers on one bb.py pool.

Each site was built once, in its own store, sharing nothing with the others except the prompts and beyondlive. So nothing written for one site could be reused in another, and this pilot says nothing about reuse or propagation. Each site was graded five times by a Playwright copy of the benchmark’s browser tester, with Mistral as the grading model.

Results. Scores are tests passed over tests, per site, averaged over the five gradings. Other figures are medians per site.

	A, files	B, bb.py	C, bb.py with architects
Score	78.2%	75.9%	72.3%
Against A: wins, losses, ties		6, 13, 1	5, 14, 1
Against A: mean difference, 95% interval		−2.3 [−5.9, +1.2]	−5.9 [−11.8, 0.0]
Agents	1	1	7
Output tokens	53k	37k	112k
Cache reads	1.3M	2.1M	7.9M
Tool calls	20	30	120
Working time	7.5 min	6.1 min	14.6 min
Functions	57	45	57
Lines per function, mean and max	22, 142	27, 133	34, 223
Cloned lines	34	25	47
Duplicated constants	65	55	95

C against B: 8 wins, 9 losses, 3 ties, −3.6 points [−10.8, +3.6].

Scores. B did about as well as A: the interval runs from six points below to one above. C did worse than A, at the edge of significance, and cost far more: seven agents, twice the output, six times the cache reads and twice the working time. The grader is noisy: on average, the best and worst of a test's five

gradings differ by 0.3 on a scale of 0 to 1, which is why each test was graded five times.

Structure. B writes fewer, longer functions than A, but less cloned code and fewer duplicated constants. C duplicates more than both.

Tools. Agents barely used bb.py's map of the code. B ran `bb tree` once per site and `bb search` twice. Before writing a new function, B looked nothing up in the median site: neither the tree nor any other context. C's workers looked up context before 42% of their new functions, and consulted the tree before 10%. The difference fits a simple reading: an agent that wrote every function in this session finds the map in its own transcript, while C's workers did not write each other's functions and have to look. Even so, C duplicates the most, so looking up context was not enough to prevent rewriting. The tree should matter most when that memory is gone: in a new session, for another agent, or for a person.

Expectations, written before the run.

- B scores within a few points of A: held;
- C shows less duplication than B, because its architect plans the functions first: failed, C duplicates more;
- B and C use `bb tree` before writing new functions: failed, because the transcript already serves as the map;
- C needs fewer interventions than in the earlier run: held, C needed no coordination help. The only interventions were one relaunch in A after a mistyped prompt path, and a one-line resume sent to fourteen agents, across all three conditions, after an account usage limit stopped them.

Rule breaks. None in A. Two sites in B. In C, a median of two per site, mostly functions saved with shell loops.

What it shows. One agent writing through bb.py builds about as well as one writing a file, a little faster, with fewer output tokens but more re-reading. Splitting the work among an architect and workers costs much more and did worse. And the tools that should give agents a map of the code went almost unused.

What it does not show. That bb.py is better; anything about reuse or propagation; anything about people. The human test decides, and it has not run.

Annex 2: the same pilot with Mistral Large 4

The same 20 tasks, the same three conditions and the same instructions, run with Mistral Vibe, Mistral's coding agent for the terminal, and Mistral Large 4 as its model. The grader is the same Playwright copy of the benchmark's tester, now with Mistral Large 4 as its grading model, and three gradings per test instead of five. Because the harness, the model and the grader all changed, scores in this annex can be compared with each other, not with the first annex.

	A, files	B, bb.py	C, bb.py with architects
Score	75.6%	72.6%	70.7%
Against A: wins, losses, ties		7, 12, 1	6, 12, 2
Against A: mean		−2.9 [−11.5, +5.7]	−4.9 [−11.0, +1.3]

	A, files	B, bb.py	C, bb.py with architects
difference, 95% interval			
Agents	1	1	7
Output tokens	107k	98k	254k
Cache reads	3.4M	3.5M	10.2M
Tool calls	56	53	276
Working time	17.3 min	13.9 min	16.6 min
Functions	41	45	49
Lines per function, mean and max	22, 104	18, 105	20, 112
Complexity per function, mean	4.8	3.6	3.8
Total lines	1092	801	904
Cloned lines	9	26	10
Duplicated constants	39	37	54

C against B: 6 wins, 9 losses, 5 ties, −1.9 points [−11.1, +7.2].

Scores. The order is the same as with Opus: A, then B, then C, and no difference is significant. B's interval is wide because of one site, train_001104, where B passed 1 test in 6 against A's 4.5. Without it, B against A is −0.1 points [−6.9, +6.6]. The three gradings of a test differ by 0.17 to 0.25 on a scale of 0 to 1, less than in the first annex, but with three gradings instead of five and a different grading model, the two spreads cannot be compared.

Cost. Mistral wrote about twice the output tokens Opus did, in every condition. B again cost no more than A and was the fastest. C used seven agents, 2.4 times A's output and five times its tool calls, but its working time was close to A's.

Structure. Here bb.py pushed the other way from the first annex: B and C wrote shorter, simpler functions than A, and less code overall. But B cloned more code than A, 26 lines against 9.

Tools. Mistral's workers in C used bb.py's tools more than Opus's did: ten `bb tree` calls, eight `bb caller` and twenty `bb search` per site. Before writing a new function, B consulted the tree 6% of the time, C almost never.

Rule breaks and interventions. No rule breaks in any condition. One intervention: a writer in A exited and was relaunched.

What it adds. The first annex's result holds with another model and another grader: one agent writing through bb.py builds about as well as one writing a file, at no extra cost. Architects and workers still did not help. What changes with the model is the shape of the code: with Mistral, bb.py gave smaller, simpler functions, and with Opus it gave fewer, longer ones. So the structure bb.py produces depends on the agent using it as much as on bb.py itself.

dx - 9 - The story keeps repeating

2026 / 11 / 07 · https://hyper.dev/2026/dx-9-the-story-keeps-repeating/

Expand these in your own agent, I publish mine next week.

Seeds

0. Every generation of software meets a codebase too big for its people, and calls it a new crisis;
1. Layers are zoning, not maps: they change when the terrain changes, and a map changes only by terraforming;
2. Much of our infrastructure is accretion, growing around designs that did not hold: change data capture around a database asked to do too much, a connection pooler around poor transaction management;
3. Nothing cured it before either: big rewrites mostly failed, and what helped was local, shared and temporary, containing the mess rather than curing it;
4. Agents do not change the story, they change its speed;
5. Agents organised as a team repeat it too: in my pilot, an architect with workers duplicated more, cost more and did worse than one agent alone;
6. So the question is how to make looking at the mess cheap, shared and safe enough that people change it locally, before it becomes infrastructure.

möbius - 10 - Je vous apprends la langue du feu

2026 / 11 / 08 · https://hyper.dev/2026/mobius-bb-py-v0-3-la-langue-du-feu/

bb.py v0.3.0, ou ce qui arrive le lendemain

« Bonjour » devient « bofonjofoufur ». C'est la langue du feu, un jeu d'enfants : après chaque voyelle, un *f* et la même voyelle. J'en avais fait la première démonstration de bb.py, parce qu'elle montre une chose simple : écrivez l'encodeur en français, en japonais ou en arabe, et les trois fonctions ont le même hash.

```
def encoder_langue_du_feu(texte): ...      # français
def 火の言葉に変換(文章): ...                  # japonais
def تشفير_لغة_النار(نص): ...                # arabe
```

```
$ bb add feu_fra.py@fra
Hash: 5fe367ff2488ad9083034991a28db92f9be781f409adc395844a36e930bf3589
$ bb add feu_jpn.py@jpn
Hash: 5fe367ff2488ad9083034991a28db92f9be781f409adc395844a36e930bf3589
$ bb add feu_ara.py@ara
Hash: 5fe367ff2488ad9083034991a28db92f9be781f409adc395844a36e930bf3589
```

Personne n'a traduit personne. Les noms et la documentation restent dans leur langue, le hash ne regarde que la logique. Les chaînes, elles, restent françaises : traduire les chaînes, c'est de l'i18n, pas du code multilingue.

Mais un programme ne vit pas une fois pour toutes. Voici ce que fait la v0.3 le lendemain.

La règle que j'avais codée n'était pas celle que je connais

Le jeu que je connais traite « ou » et « oi » comme une seule voyelle : *mou* devient *moufou*, *moi* devient *moifoi*. Mon encodeur, lui, double chaque lettre voyelle une à une. J'écris donc les trois cas dans un fichier d'édition, sous forme de checks, et j'essaie de sauvegarder l'ancien code :

```
$ bb edit encoder_langue_du_feu@fra feu.md -m "Ajouter les checks"
FAIL check_feu_moi: returned False ('mofoifi' == 'moifoi')
FAIL check_feu_mou: returned False ('mofoufu' == 'moufou')
FAIL check_feu_accent: returned False ('été' == 'éfétéfé')
3 checks: 3 failed
Not saved: fix feu.md and save it again.
```

Les checks font partie de l'identité d'une fonction, et un check qui échoue empêche de sauvegarder. Je corrige donc le code : une liste de groupes ordonnée par priorité (« eau », « ou », « oi »… avant « o » et « u »), puis les voyelles accentuées, puis les voyelles simples.

```
$ bb edit encoder_langue_du_feu@fra feu.md -m "Groupes de voyelles ordonnés, voyelles
accentuées"
PASS check_feu_moi
PASS check_feu_mou
PASS check_feu_accent
3 checks: 3 passed
Parent: 5fe367ff2488ad9083034991a28db92f9be781f409adc395844a36e930bf3589
Hash: bb860da6d956ba2dfbcc6a19d7fb04a225a1a2196050ae1ea8eb5b0ffb60c268
```

Nouvelle logique, nouveau hash. L'ancienne version n'est pas effacée : elle est le parent de la nouvelle. Aucune des deux n'est « la vraie », ce sont deux règles du même jeu.

Le japonais et l'arabe attendent

J'ai changé la version française. Que deviennent les deux autres ?

```
$ bb search --outdated
Name: تشفير_لغة_النار (ara)
Status: ..., newer version in fra, not yet translated to ara, newer version in fra, not
yet translated to jpn
```

Ils n'ont rien perdu et rien cassé : ils restent sur leur version, et bb dit simplement qu'une traduction est attendue. Dans la v0.3, une modification dans une langue ne force personne dans les autres, elle laisse une dette visible.

Quelqu'un qui lit le japonais la paie, en donnant les noms dans sa langue. Le code ne change pas, seuls les noms changent :

```
$ bb translate encoder_langue_du_feu@fra jpn traduction.md
- encoder_langue_du_feu -> 火の言葉に変換
- texte -> 文章
- voyelles -> 母音一覧
- voyelle -> 母音
...
$ bb translate encoder_langue_du_feu@fra jpn traduction.md
Translation:
bb860da6d956ba2dfbcc6a19d7fb04a225a1a2196050ae1ea8eb5b0ffb60c268@jpn@6c9f9c74…
Name: 火の言葉に変換@jpn
```

Même geste pour l'arabe. Ensuite :

```
$ bb search --outdated
No matches found: nothing outdated
```

Trois noms, un hash, et une histoire

```
$ bb log 火の言葉に変換@jpn
Step: 5057cc25…
Translate: 火の言葉に変換@jpn bb860da6…
From: 火の言葉に変換@jpn 5fe367ff…

Step: 94551f4e…
Translate: 火の言葉に変換@jpn 5fe367ff…
```

Le nom japonais a maintenant deux versions, et chaque pas dit d'où il vient. Les trois langues ont convergé de nouveau sur `bb860da6`…. La convergence est détectée, la divergence est préservée : c'est toute la promesse du projet, jouée sur un jeu d'enfants.

Ce qui change pour vous

La v0.3.0 couvre 109 commits depuis la v0.1.0. Deux ruptures : le stockage (`bb migrate` convertit les anciens pools) et les imports (`from bb.pool import x` devient `import bb` puis `x = bb.resolve("x")`). `@check` et `bb commit` ont disparu : les checks vivent dans le fichier d'édition, et partager un pool relève de git. Les notes complètes sont sur la page de la release : https://github.com/amirouche/bb.py/releases/tag/v0.3.0

La prochaine fois, je vous apprends la langue du meuh. 🐮

purview - 46 - 2026, week 46

2026 / 11 / 09 · https://hyper.dev/2026/purview-2026-week-46/

Purview is a weekly review of blog posts I find on the web and how they relate to my work. It is also a way to shout out interesting work, even when it sits outside my comfort zone.

This week's thread is understanding: what it takes to keep software small enough to grasp, and what happens to that grasp when code costs almost nothing.

A Conversation with Paul Masurel, Creator of Tantivy – ParadeDB

Paul Masurel, who wrote Tantivy as a Rust take on Lucene, was interviewed by ParadeDB in April 2026, a company that runs Tantivy inside PostgreSQL.

Tantivy is a library, not a service, and I think that is the future of search: not huge deployments, but situated databases that people first build and then query the way they want.

Link: https://www.paradedb.com/blog/tantivy-interview

Source control at agent scale – OpenAI, Anthropic, and jj

Four moves in a few months, one cause: coding agents push far more code than version control built for humans can absorb.

OpenAI kept Git and rebuilt the server. Pando is a Git-compatible backend written in Rust, designed by one engineer and built with Codex in ten weeks. It has served the monorepo since April, cut median push latency by nearly 80%, and engineers and agents still use the plain Git client.

Anthropic left Git. Merges to its monorepo went from about 2,500 a day in May to more than 30,000 in September. A team of seven built Scion, a centralized system that assumes agents are always online, and migrated in five months. It is not open source, and Anthropic says it is no replacement for Git in general use.

GitHub, the host for everyone else, also kept Git and rebuilt underneath. Its new storage layer, announced on November 5, keeps Git packs in Azure Blob Storage, references in a dedicated replicated log, and serves traffic from disposable replicas that hold only cached state. In internal tests it took 5,000 concurrent agents and 7,200 pushes per second on one repository. It is in private preview, with a wider rollout planned for early 2027.

Martin von Zweigbergk, who created Jujutsu (jj), joined East River Source Control as CTO in September, after 15 years at Google. In his view jj fixes the side of version control that runs on your laptop, but the server still speaks Git, and the storage layer is what must change next. jj stays open source under Apache 2.0, with Martin as a core maintainer.

Why it matters to me:

- while there is a strong push for situated and ad-hoc software, big players are aiming and building big;
- Möbius offers a simple way to coordinate change at scale: merging two separate branches costs nothing, even when they share common idioms at the leaves, and integration remains as it does with Git when overlapping changes merge, so Möbius reaches a working codebase faster. The other side of the coin is near-duplicate functions, which call for a deduplication and factoring pass that is not

implemented yet, and readability and maintenance stay as unresolved as anywhere else;
- Möbius is a Unison scion, built on content-identified functions with a dedicated VCS, which leads me to wonder what the Unison team will answer.

Links:

- Scaling Git at OpenAI;
- Redesigning source control at Anthropic;
- East River Source Control Names Jujutsu Creator Martin von Zweigbergk Chief Technology Officer;
- Inside GitHub's new Git storage architecture.

Curiosity Traps – The Autodidacts

Curiositry turns the honeypot around: instead of luring attackers, a curiosity trap is a small hook that rewards close attention and draws in like-minded people. Examples range from the turtle in the Wait But Why footer, which leads down a chain of nested pages, to footnotes, inside jokes, and job offers hidden in a browser console.

The remedy is mostly patience: publish often, write from concrete experience, make it easy to reply, and layer the text so the main point stays fast for hurried readers while the details reward the attentive ones.

All the footer is a trap. The only missing piece is a blogroll, and I am working on it. It feels premature to link to the blogs of people I have not had a conversation with beforehand.

Link: https://www.autodidacts.io/curiosity-traps/

When two people meet, it can spark and generate interesting points of view

Pavel Uvarov wrote Schevana, a functional desktop in Chez Scheme. He calls it a toy, a happy place for Lisp enthusiasts: the desktop is just a function of the environment, drawn by the CPU in a single thread.

dharmatech picked it up and built upon it, with a pull request that adds a flock of boids and a window to edit its rules. He also made a video where he explains that AI offers the opportunity to visit or revisit topics, new and old, more quickly and in a way that is more fun.

One person shares the work of the other and builds upon it. That opens the question of what AI is for, and whether there is still room for experimenting and improving upon the state of the art.

Why it matters to me:

- it is the same experience I had with seed, my interpreter for a dialect of Scheme in the Kernel family;
- AI helped me bridge the gap between the vision and the actual code, and made the dream partly a reality.

Links:

- Schevana;
- Boids: a flock, and a window for its rules;
- dharmatech's video.

Futamura projections – Veit Heller

Veit Heller is writing a series that explains Futamura projections, with a first and a second projection so far.

Futamura projections bridge the gap between interpreters and compilers. Given an interpreter and a specializer, they

promise fast, ad-hoc compilation strategies at a lower cost than writing a compiler by hand.

Why it matters to me:

- it is the road I am taking with seed.

Links:

- A First Futamura Projection;
- A Second Futamura Projection.

Artificial intelligence is not intelligence augmentation

All seven texts agree that understanding has become the scarce thing. A program is a theory held by people, not its text, cognitive debt lives in heads, and generation now outpaces understanding. The loss hides itself: satisfaction rises while skill falls, and AI users overrate their skills. What decides is the relationship to the tool, not the tool.

They split on where understanding can live, and on what to do. Storey and VirtusLab keep it in people's heads and answer by slowing down. Kerr notes that agents cannot know by making but excel at knowing by testing, so she builds verification and a context for agents. My essay AI without IA puts part of it in the environment. On the stakes, Hicks shows that it is a matter of identity, and Jeffries abstains where Kerr uses AI despite its harm.

Kerr's list of vexations leaves out two techniques. Mutation testing tests the tests themselves, which matters most when an agent writes both the code and its tests. Deterministic simulation testing runs a system under a controlled clock and scheduler, so that any failure can be replayed exactly. I told the

story of a small mutation testing tool in Reading mutation.py: a story.

Where Naur speaks of theory, I like to speak of the spirit of the code, as opposed to its letter: something you have to make your own when you approach a new codebase. It is a more elegant way to say: when in Rome, do as the Romans do. Going through an AI tool, we lose that spirit, or never acquire it.

Links:

- How Generative and Agentic AI Shift Concern from Technical Debt to Cognitive Debt;
- AI coding creates two kinds of debt. You're only measuring one;
- How AI coding tools silently erode developer understanding;
- Who are we Now?;
- AI Skill Threat & contest cultures on software teams;
- 'AI', etc.;
- AI without IA.

The Lisp curse is coming to your ecosystem

Tim Zöller, writing from Java, takes up the Lisp curse: Lisp makes custom solutions so easy that everyone builds their own, and no shared library forms. He fears LLMs bring it to ecosystems like Java's. If generating code looks cheaper than adopting a library, fewer people use libraries, maintainers get less feedback, fewer reusable modules get shared, and software development becomes a lonelier task.

I think the diagnosis is right and the name is wrong. In Rudolf Winestock's essay, the curse is social: extending Lisp is so easy that lone hackers each build their own 80% solution, and the

community never converges. But the lone hacker at least understood the code. What Zöller describes is code that nobody wrote and nobody read. That is not Lisp's curse, that is artificial intelligence without intelligence augmentation, the thread of this issue.

The ordered key-value store taught me the same lesson from the data side. OKVS is the factorization of what you need to store data: ordered keys, bytes, ranges, transactions. It is so powerful that two people never need to agree on code; each rebuilds secondary indexes or pagination in an evening. They do need to agree on the key encoding, because bytes do not explain themselves and data outlives the code that wrote it. FoundationDB's tuple layer is first of all an encoding, written down as a spec, which is why bindings in many languages agree on the bytes without sharing code. SRFI-167 and 168 were my attempt at the same level.

So the curse is not having too much power. It is failing to separate what must be shared, the spec, from what one can afford to rewrite, the code. LLMs threaten that separation from the other side, by making code so cheap that nobody bothers to write the spec.

Links:

- The Lisp curse is coming to your ecosystem, thanks to LLMs, by Tim Zöller;
- HN discussion;
- The Lisp Curse, by Rudolf Winestock.

Today I Learned

Today I learned a word from Jessica Kerr, who signs her site "symmathecist, in the medium of code". A symmathesy, a word

coined by Nora Bateson, is a learning system made of learning parts. A symmathecist is an active participant in one, who works to increase learning within the system and between the system and its environment.

For Kerr, a software team is a symmathesy: people, running software, and tools all learn from each other. People learn from errors and stored data, the software changes as people change it, and none of the parts stays fixed.

Links:

- jessitron.com;
- Symmathecist (n.).

purview - 47 - 2026, week 47

2026 / 11 / 16 · https://hyper.dev/2026/purview-2026-week-47/

Purview is a weekly review of blog posts I find on the web and how they relate to my work. It is also a way to shout out interesting work, even when it sits outside my comfort zone.

Do code maps make agents cheaper? — Brice Bulgarelli, and Sepidband, Shin, Wang, Hemmati

Brice Bulgarelli built Seam, a deterministic tool that follows call stacks across languages, and measured it on Keycloak with Claude Code. Seam cut tool calls by about 36% and still raised the bill by about 1.4x compared to grep. Even an idealized tool replaying perfect answers cost 1.13x to 1.26x when it replaced search, and 0.91x when the agent kept grep beside it. A tool that returns only the answer, about 500 tokens, cost 0.45x grep. The agent kept verifying for about ten more calls after it had the answer, which he calls the verification floor. This is one task on one repository, five rounds per arm, and on Sonnet the floor was smaller.

He also published a short French version, Pourquoi un outil d'agent peut coûter plus cher.

Link: https://blog.bulgarelli.dev/papers/cout-outil-structurel/

How Should Agents See Code?, a York University study, looks at five structured views of a repository (AST, call graph, dependencies, docs, hybrid), exposed to Claude Code with Haiku 4.5 as MCP servers. They help most when finding the code is the bottleneck, as in fault localization, where tokens dropped 25% to 39%. They fade when the hard part is writing the patch or the test.

Link: https://arxiv.org/html/2611.07286v1

The York study finds structured views save tokens when locating code is the hard part. It counts total tokens. An earlier study that Brice builds on (Token Reduction Is Not Cost Reduction, by Weinberger and Hozez, arXiv 2607.12161) showed that fewer tokens is not a smaller bill, because cache writes cost more. What Brice adds is the verification floor, and a way to price a perfect tool before building one. The two answer different questions, and together they say what to measure: the bill, not the token count.

My next step: build a large application from scratch, over several iterations, and test whether bb.py reproduces the results of these two studies.

Opinion is also bubbling up about using alternative code representations to help agents work on code, as in Jason Davenport's Why Git Needs Help: Moving from Files to Systems for Source Control.

Multi-step SWE – Klavis AI

A benchmark of 69 tasks and 574 dependent steps across 9 languages, asking whether coding agents can keep building on their own code. The strongest of 9 models solves 18.5% of tasks, and 38 tasks are solved by no model.

Link: https://multistepswe.com/

Cognitive Tours – Margaret-Anne Storey

Storey names two debts next to technical debt: cognitive debt, when no one on the team can explain how the system works anymore, and intent debt, when the goals and rationale are

missing. Her prototype lets humans write curated tours through needs, decisions, code and tests, while AI only audits them. The friction of writing a tour is the point. There is no evaluation yet.

Link: https://arxiv.org/abs/2611.08456

Who Can Evaluate AI Independently? – Adan et al.

The authors propose PAAI (Power, Access, Adequacy, Independence) to ask who controls the infrastructure behind an AI evaluation. Applied to twelve African AI evaluation projects, it shows African actors supplying data, language expertise and task design while others keep the infrastructure. Taking part is not the same as having authority.

Link: https://arxiv.org/abs/2611.08134

Self-distillation inherits the reversal curse – Mojeek

Mojeek, one of the few search engines running its own independent index, published a small controlled experiment on 5 November. They self-distilled Qwen2.5-7B on invented facts about films, directors and studios, then checked whether the model could answer in the reverse direction. It learned the facts in the direction they were given and almost nothing else, so distillation inherits the reversal curse. Their remedy: have the model write each fact from every cue a later question could start from, and test reversed and multi-step questions, not just recall.

Why it matters to me:

- writing each fact from every cue is what an ordered key-value store does when it indexes a tuple under all its

permutations, which is the design behind the generic tuple store aka. nstore, standardized in SRFI-168;
- my thesis is to keep facts out of the weights, put them in a store that answers in every direction, and let the model handle the language;
- my own guess, which the post does not test (all of it is in English), is that the curse bites hardest on non-English names and facts, such as Arabic or Amazigh entities and transliterations.

Link: https://blog.mojeek.com/2026/11/self-distillation-inherits-the-reversal-curse.html

Let Over Lambda – Doug Hoyte

A shout-out to *Let Over Lambda—50 Years of Lisp* by Doug Hoyte, a book about macros and the advanced features of Common Lisp, first published in 2008. Its 2026 update says the entire book is now free to read online, in HTML and PDF. The print edition runs about 380 pages, and that is a lot of effort to put into a book on macros.

I prefer fexprs and Kernel, and I don't know Common Lisp. I will probably read it after SICP!

Link: https://letoverlambda.com/

Search without an account: pay per query over HTTP 402 – six search API prices compared

Three search APIs now sell by the query, with no account and no API key. The agent sends a request and gets back HTTP 402 with a price. It signs a USDC payment, retries, and receives the results with a receipt. It runs on x402, an open protocol. Brave joined on 12 November.

From the lowest price per query to the highest: Mojeek asks £0.002 to £0.003 for raw results from its own index, with no x402 and at most 10 results per request on the cheapest plan. Exa starts at $0.004, but only in its lightest mode, and its deep modes reach $0.015. Brave is at $0.005, paid in USDC on Base only for now. Tavily is at $0.01 on the same rail, twice Brave's price. Kagi is at $0.012 and still needs an account and an API key. Bing is at $0.014 and no longer sells raw results: it returns summarized grounding, only inside Azure. The trade-off runs in one direction. The cheaper offers give you raw results you can build on, and ask you to accept a smaller index or a crypto wallet. The more expensive ones give you ranking quality or an answer, and tie you to an account or a cloud.

Links:

- https://brave.com/blog/search-api-agentic-payments/;
- https://www.tavily.com/blog/tavily-on-x402;
- https://exa.ai/docs/integrations/payments/x402/quickstart.

Today I Learned

A token added to an agent's context has to save about 20 tokens of re-reading to pay for itself, because writing to the prompt cache costs 20 times more than reading from it. I learned it from Brice Bulgarelli's study above.

purview - 48 - 2026, week 48

2026 / 11 / 23 · https://hyper.dev/2026/purview-2026-week-48/

Purview is a weekly review of blog posts I find on the web and how they relate to my work. It is also a way to shout out interesting work, even when it sits outside my comfort zone.

Four links this week. Two of them, hashloom and JoyFrontBench, share one idea from opposite ends: write the contract before the code, then judge the code against the contract. The other is stipple, a tool idea to escalate from locally available desktop tooling to cloud frontier capabilities when necessary.

multilingual: keywords in your language, one program underneath

John Samuel's *multilingual* (on PyPI as `multilingualprogramming`, version 0.8.2, June 2026, GPL-3.0-or-later) is a programming language whose keywords can be written in several human languages. Each language has a frontend that maps its keywords to a shared core; all frontends share one parser and one AST, with a semantic IR planned; the compiler emits Python or WAT/WASM. A module written in French, with `soit` and `retour`, can be imported from a program written in English.

The author is candid about where it stands: the semantic core is thinner than the planned Core 1.0, and some localized keyword sets read less naturally than intended.

Why it matters to me. It works on the other layer from bb.py. multilingual localizes the keywords and leaves identifiers as written, so `incremente` is still `incremente` when called from English. bb.py keeps Python keywords and normalizes identifiers, so

`additionner` and `add` land on the same hash. Two halves of the same problem. I should write to John Samuel.

Links:

- https://pypi.org/project/multilingualprogramming/;
- https://github.com/johnsamuelwrites/multilingual.

hashloom: contracts are warp, code is weft

hashloom, by davet47, is an MCP server that treats a software unit as a content-addressed contract rather than a file. A contract is a small YAML spec: signature, invariants, examples, names of dependencies. The contract stays fixed; the implementation behind it is meant to be regenerable.

What the hashes are used for:

- the implementation hash is SHA-256 over the normalized AST, so formatting and comments do not change it;
- a green test run is cached under a key made of the contract, implementation, test source, toolchain and dependency-contract hashes, and failures are never cached;
- a contract change reports its transitive dependents by hash;
- an agent gets a packet of about 300 tokens (contract, dependency signatures, callers) instead of whole files.

The core is Python with SQLite for the local store; adapters for Go, TypeScript, Java and C# use each language's own tooling. Two stars, 95 commits, Apache-2.0. The README reports a 5.4x token reduction on a 20-contract sample, self-measured, with the author's own caveats.

The hashes drive cache invalidation and dependency tracking; they do not store or retrieve code by content. Content addressing as a verification cache, not as a code pool.

Why it matters to me. "Contract fixed, code regenerable" is close to where bb.py is heading with asserts, and later with checking equivalence before accepting a function replacement.

Link: https://github.com/davet47/hashloom

stipple: the cloud as last resort

stipple, by thatz0083, is a routing layer in front of cloud models, described as a local-first layer for LLM apps. For each question it walks about ten ordered checks: commands, a calculator, saved preferences, installed Packs, a store search, a cache of earlier cloud answers. Only when nothing matches does it call a cloud model. Cloud answers are cached automatically; preferences learned in conversation live in the browser.

It is a prototype. It runs entirely in the browser, the README says the cloud calls are simulated, the on-device model tier (WebLLM) is disabled, and similar-question matching is off by default, so only exact repeats hit the cache. MIT, one star, public since June 2026, renamed from DOTS in September.

What I take from it is the order of the checks, not the code: the frontier model as the last resort, after everything a single machine can answer by itself.

Link: https://github.com/thatz0083/stipple

JoyFrontBench: task scores are not application success

JoyFrontBench: Evaluating Front-End Coding Agents through Goal-Oriented Acceptance Contracts (Qiao et al., arXiv 2611.14952, 12 November 2026) starts from the observation that front-end coding agents can post high task scores while failing the workflows that matter. Its answer is acceptance contracts written before generation: core tasks, mandatory criteria, a shared completion threshold.

- 100 problems, 764 tasks, 2,596 atomic criteria;
- a core task passes only when all its mandatory criteria hold and the threshold is met;
- application success (ASR) requires every core task to pass on the same artifact in a single run;
- five models, two harnesses (OpenHands and OpenCode), 70 problems for test and 30 public for validation.

Under OpenHands the mean average task score is 87.69% while the mean application success rate is 51.52%. Rankings move between harnesses, and ASR moves between runs.

Why it matters to me. A candidate for the bb.py benchmarks, next to WebGen-Bench: same front-end ground, and it measures whole-application success instead of an average that hides failures. To check: whether the 30-problem public split can be downloaded and run.

Link: https://arxiv.org/abs/2611.14952

Today I Learned

You can hash the normalized AST to key a verification cache without making the AST the source of truth. hashloom keeps YAML contracts as the stable object and uses implementation hashes only to decide when tests must run again.

möbius - 11 - bb.py v0.6.0, qui a écrit ça, et qui l'a lu

2026 / 11 / 26 · https://hyper.dev/2026/mobius-bb-py-v0-6-0-qui-a-ecrit-ca/

La confiance qu'on peut vérifier

Bob ne lancera pas la fonction de Dia. Il n'a rien contre elle : il ne la connaît pas.

Dia a écrit `show_double`, elle l'a relue, et elle a répondu oui à trois questions : sécurité, exactitude, lisibilité. bb a publié son verdict dans le dépôt partagé, signé avec sa clé SSH. Bob ouvre ce même dépôt et demande :

```
$ bb reviews show_double@eng
Function: 622a6c87530d8dac589db37c98dc6608cff82ae935ef0805d5d354ac4b731ee8
You: not reviewed
untrusted  approve  Dia <dia@example.org>  2026-11-24T09:05:10Z  Read it
    acknowledged: correctness, readability, security
    read as show_double@eng (mapping 53a63e33c5df...)
$ bb run show_double@eng 3
Error: Refusing to run code you have not reviewed (1 function(s) to approve).
Review it first: bb review
622a6c87530d8dac589db37c98dc6608cff82ae935ef0805d5d354ac4b731ee8
```

La signature est valide, mais la clé est celle d'une inconnue : `untrusted`. N'importe qui ayant le droit de commiter peut écrire un fichier dans ce répertoire, alors une signature ne compte pour Bob que s'il a choisi son auteur. Dia lui a envoyé sa clé publique par un canal qu'ils partagent déjà, hors du dépôt. Il l'importe de la liste partagée, une entrée à la fois, et compare la clé avant de répondre :

```
$ bb trust import
dia@example.org ssh-ed25519
```

```
AAAAC3NzaC1lZDI1NTE5AAAAIDiaWroteShowDoubleAndReadItBeforeBobDidOk42
Trust this peer? [y/n]: y
Trusted: dia@example.org
Trusted peers: 1
```

Cette question est la porte. La liste partagée contient des candidats, pas des pairs : quiconque peut commiter peut y ajouter une ligne. Le `y` doit donc rester le vôtre ; ne branchez jamais `yes` dessus.

La même ligne se lit maintenant autrement, et la fonction tourne :

```
$ bb reviews show_double@eng | sed -n 3p
trusted  approve  Dia <dia@example.org>  2026-11-24T09:05:10Z  Read it
$ bb run show_double@eng 3
6
```

Et si quelqu'un modifie un mot de la review de Dia dans le dépôt ?

```
$ sed -i 's/Read it/Never read it/' $BB_DIRECTORY/review/*/*/*/*.json
$ bb reviews show_double@eng | sed -n 3p
invalid  approve  Dia <dia@example.org>  2026-11-24T09:05:10Z  Never read it
$ bb run show_double@eng 3
Error: Refusing to run code you have not reviewed (1 function(s) to approve).
Review it first: bb review
622a6c87530d8dac589db37c98dc6608cff82ae935ef0805d5d354ac4b731ee8
```

Une ligne, trois états : `untrusted`, `trusted`, `invalid`. C'est toute la v0.6.0 en une scène. Les signatures disent qui parle. La confiance dit qui vous croyez. Les reviews, les vôtres ou celles des pairs que vous avez choisis, décident de ce qui peut tourner.

Faire confiance à Dia ne vaut pas que pour cette fonction : toutes ses approbations signées qui répondent aux trois questions comptent désormais. Le rejet d'un pair de confiance

bloque l'approbation d'un autre, votre propre verdict passe avant, et `bb trust remove` retire un pair.

Attention : bb n'a pas de bac à sable. Une fonction approuvée s'exécute avec vos droits, la review est la seule barrière. Faire confiance à Dia, c'est la laisser décider, en partie, de ce qui tourne sur votre machine (#91).

Avant de mettre à jour : il n'y a aucune garantie de compatibilité ascendante ni de migration d'ici la v1. Un pool écrit par la v0.5.0 peut ne pas se charger avec la v0.6.0, et les versions 0.x suivantes ne promettront pas davantage ; `bb migrate` a disparu pour cette raison. Gardez votre pool dans git, et attendez-vous à tout réenregistrer.

Pourquoi maintenant

La v0.5.0 avait rendu les checks honnêtes : un check doit `assert`, sinon il ne peut jamais échouer. La v0.6.0 s'occupe de la question suivante : qui a écrit ce que je m'apprête à lancer, et qui l'a lu ?

Ce qui est signé, ce qui est daté

Avec `bb whoami signing-key`, tout ce que vous enregistrez est signé par `ssh-keygen` : les fonctions, les mappings (les noms et la documentation, dans une langue), les étapes de lignée (de quelle version vient la suivante), les worklogs et les reviews. `bb log` affiche `Signed:`, et `bb show` dit, au-dessus du code, ce qui se porte garant de la fonction. Un échec de signature n'empêche jamais d'enregistrer : bb prévient, puis enregistre sans signature.

La date d'un enregistrement est ce que dit son auteur. `bb anchor` demande une preuve OpenTimestamps pour chaque fichier, qui atteste qu'il existait au plus tard à une date donnée, et

prend un instantané du pool pour qu'un enregistrement disparu plus tard se voie. C'est la seule commande de bb qui touche le réseau.

Les reviews sont liées au hash. `bb run`, `bb show`, `bb edit`, `bb compile`, `bb update` et `bb refactor` recalculent le hash de ce qu'ils lancent ou réécrivent : un objet remplacé sous un hash approuvé est refusé. Ces vérifications viennent d'une relecture adversariale de la branche par Claude, qui a fermé toute une série de failles, de la comparaison des clés de review jusqu'à `bb update` exécutant les checks d'un code que personne n'avait approuvé.

Aussi dans cette version

- Plusieurs fonctions peuvent répondre à un même nom. `bb search NOM` dit comment elles se rapportent, et `--compare` lance chacune contre les checks de l'autre.
- Dans le code, `bb.resolve("nom")` désigne un nom, `bb.ref("<hash>")` épingle une version exacte. Un nom ambigu se règle une fois, au terminal ou avec `--pick NOM=HASH` ; les agents passent `--no-ask`.
- `bb scratchpad --show` et `bb edit --show` disent ce qu'enregistrer ferait.
- `bb search --pinned` et `--outdated` trouvent les références épinglées et celles qui sont en retard, et `bb update` les fait suivre.
- Python 3.11 est de nouveau pris en charge.

Venez en discuter

Quatre issues sont encore ouvertes dans leur conception, et c'est là que vos avis changent quelque chose. Un commentaire, une contre-proposition ou une nouvelle issue (« c'est faux parce que… ») sont tous bienvenus :

- `bb search --near` (#95) : chercher par l'exemple, classer le pool par proximité avec une fonction, d'une langue à l'autre, en disant pourquoi. Comment mesurer « proche » ?
- Checks symboliques (#94) : un check qui renvoie une propriété, prouvée par Z3 pour toutes les entrées d'un domaine, ou un contre-exemple. J'aimerais aussi savoir comment suggérer des tests de propriétés quand la preuve est hors de portée : des entrées générées, ou un petit coup de coude quand un check ressemble à une propriété. Répondez sur #94, ou ouvrez une nouvelle issue si le sujet mérite son propre fil.
- Typage par prédicats (#93) : un type est un prédicat stocké dans le pool, les `assert` d'entrée et de sortie disent le domaine et le résultat, les signatures restent nues.
- Bac à sable (#91) : dans quoi `bb run`, les checks et la sortie de `bb compile` devraient-ils tourner ?

Pour commencer petit : #79 (auditer les messages d'erreur), #77 (relire et traduire le README dans votre langue), #76 (une quatrième langue dans la transcription « langue du feu ») et #81 (une nouvelle transcription, dans votre langue).

Merci

À Chirag Jangir, qui a fait tourner les tests sur macOS dans la CI (#84) et écrit le guide des quinze premières minutes de CONTRIBUTING (#85). À Niveditha Nagappan, qui a écrit le README en tamoul (#92).

Et aux étoiles : tkersey, braillescreen, amazerisme, lojikil (son blog) et shivanshjain456. Une étoile dit à un projet d'une seule personne que quelqu'un le lit.

Si vous voulez ouvrir une issue, en commenter une ou envoyer une pull request, dans n'importe quelle langue, vous êtes les bienvenus.

Les notes de version sont sur la page de la release, et l'installation se fait avec `pip install beyondbabel==0.6.0`.

Bob ne connaissait pas Dia ; il sait maintenant ce qu'elle dit avoir lu, et il a choisi de la croire. Au-delà de Babel, il y a du lien, pas seulement des hyperliens : une merveille qui détecte nos convergences, laisse vivre nos divergences et garde trace de ce que nous avons accepté de partager.

Colophon

· /discourse/ · /acknowledgements/ · /traffic/ · /scheme/ · /ai/ · /book/ ·

🗒️ hyper.dev · 📫 hello@amirouche.dev · 🐙 github

cc-by-nc-sa — Amirouche Amazigh BOUBEKKI

---

1. **grounded entropy** (2017: “grounded topologies”, mislabeled) — let the knowledge graph grow as rich and messy as it wants, but let each connection come from a human act: a note, a move in a game, a crowd’s answer. Grounding sounds like it should reduce entropy; that tension is the point. The shape that emerges is the topology.↩

2. **cogni-games** have prior art: Luis von Ahn’s “games with a purpose” (the ESP Game, 2004, then reCAPTCHA, 2007) and Foldit (2008) already turned play into computation. Those games harvest the player’s work; cogni-games aim for the player to grow too, and open the door to people with disabilities and to art. In 2022 that became: we play, they learn, and we grow together.↩