hyper.dev

ⴰⵎⵓⵔⵓⵛ ⴰⵎⴰⵣⵉⵖ ⴱⵓⴱⴻⴽⴽⵉ

Amirouche Amazigh BOUBEKKI

- hyper.dev
- 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 - the invariant, stories from the long silence
- 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
- 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.

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.

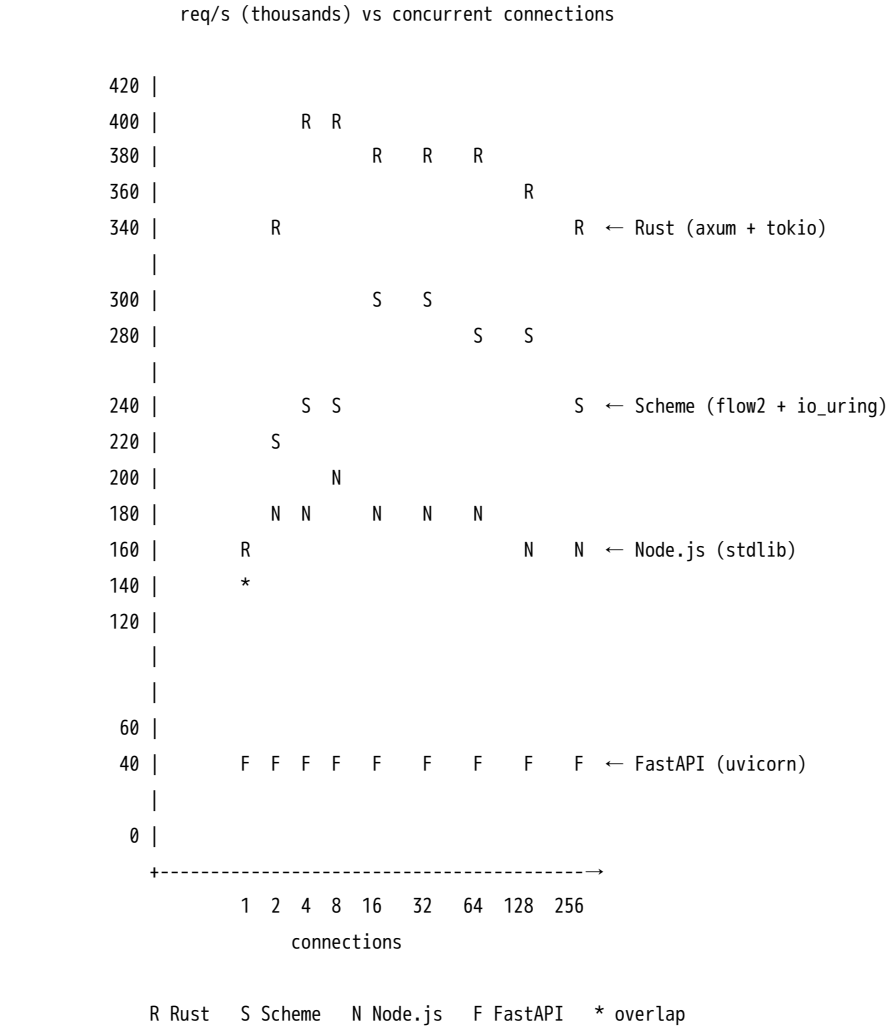
The Scheme server now runs on flow2, letloop's concurrency library, and on nothing beneath it: every read, write, timer, and accept goes through flow2's public API.

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."

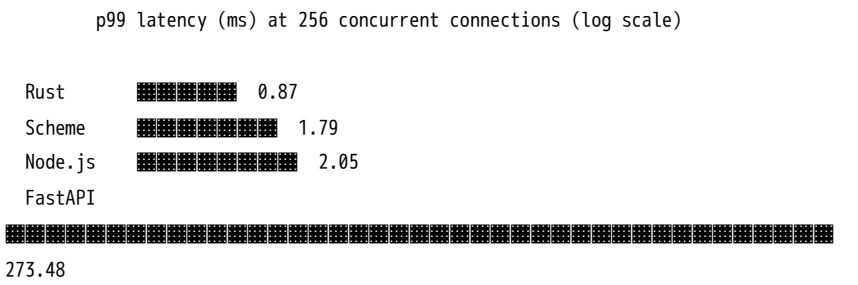
Here's how throughput scales with concurrent connections for four runtimes that represent four different async strategies:



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 - 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	🎓 Académique	2011	EN

#	Source	Type	Date	Langue
	of FLOSS in Brazil », JITP			
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

#	Source	Type	Date	Langue
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
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	🏛️ Primaire	2020	EN

#	Source	Type	Date	Langue
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
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

#	Source	Type	Date	Langue
4	D. Haney, « NPM & left-pad: Have We Forgotten How To Program? »	📝 Blog	03/2016	EN
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	📰 Presse spéc.	01/2022	EN

#	Source	Type	Date	Langue
	Highlights Software Supply Chain Issues »			
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-sponsored Elasticsearch fork, reaches 1.0 »	📰 Presse spéc.	07/2021	EN
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%	📰 Presse spéc.	01/2025	EN

#	Source	Type	Date	Langue
	of developers are unhappy »			
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
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	🏛️ Primaire	03/2025	EN

#	Source	Type	Date	Langue
	inventory »			
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

#	Source	Type	Date	Langue
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
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	📰 Presse spéc.	10/2025	EN

#	Source	Type	Date	Langue
	Statistics & Trends [2025] »			
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
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	📰 Presse gén.	05/2025	EN

#	Source	Type	Date	Langue
	supply chain relies on hidden African workers »			
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.

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.

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.

L'aube

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

Le crépuscule

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 - the invariant, stories from the long silence

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

English

01 — The Hold

We sealed the Hold at 0900 local. Eighteen months. No shore leave. The lock cycled twice, the way the procedure said, and then it stopped cycling and that was the new condition.

The Hold is a confined-environment analog: a steel cylinder buried in the side of a hill on a research site that does not appear on any commercial map. From the outside it looks like a fuel tank. From the inside it is the spaceship we will not get to fly. Same volume. Same air budget. Same compute. Same crew.

We are eight. The languages on the wall are eight. Tamazight, Wolof, French, Vietnamese, Spanish, Arabic, English, Portuguese. The wall is a piece of vinyl tape with the names written by the person who arrived first, in their handwriting, in their alphabet. Nobody asked them to do it. It is the first artifact in the Hold and it sets a tone.

The system we are testing has a rule: every name is a view onto a hash. The hash is the truth. The name is hospitality. We will spend the next eighteen months finding out whether that rule survives boredom, fatigue, an air-cycle

alarm at three in the morning, and the slow realization that nobody on the outside will be able to help us when we are no longer pretending.

Day one. Hash exists. That is enough.

The Garage Lab Hacker

Yusuf arrived at the Hold with a duffel bag, a soldering iron, and the conviction that any problem solvable in theory was solvable before breakfast. He was not an official member of the engineering team. He had been invited because he had already built a content-addressed store in a weekend, for reasons no one fully understood, and it worked.

His workspace was a corner of the maintenance bay where the smell of machine oil never quite left. He strung a cable from the structural sensor array to a Raspberry Pi cluster he had assembled from spares, and called it his testbed. The Hold's official systems team regarded him with the particular wariness reserved for people who produce results without filing tickets.

The first problem Yusuf solved was bootstrapping. A content-addressed store needs a way to register its own primitives — but those primitives are not yet in the store when you start. He wrote the primitive table on paper first, then transcribed it into a file that the evaluator could load before the store was consulted. The paper still existed, taped to the wall above his Pi cluster. The team called it the Rosetta Stone.

He did not name his combiners carefully at first. He called them things like 'q', 'qq', 'qqq'. When Amara, the Polyglot Programmer, found out, there was a brief and spirited disagreement. Yusuf's position was that names were presentation, not logic, and the hash was what mattered. Amara's position was that a system no one could read was a system no one would trust. They were both right. The naming system they designed together was the one that shipped.

On the eighteenth day of the first rotation, the Pi cluster's power supply failed. Three hours of work were in volatile memory. Yusuf retrieved it from the hash log, because every evaluation had been committed. He told no one. He just kept working.

The Polyglot Programmer

Amara spoke seven languages before the Hold, and she was not particularly impressed by systems that spoke one. The store, when she first encountered it, was English-only in its surface mappings. She filed a formal request on day three of her first rotation. The request was nine pages long and included a formal linguistic analysis of why naming systems that assumed a single human language introduced maintenance hazards.

The team lead, a quiet Swede named Erik, read all nine pages. Then he asked Amara to explain the de Bruijn normalization step in plain terms. She did. He said: 'So the hash doesn't know what language the name is in.' She said: 'Exactly. The hash is the truth. The name is the door.' He

approved the multilingual mapping work the same afternoon.

Amara added Wolof first. She had grown up in Dakar and knew that Wolof was a language with precise technical vocabulary that most software systems had never tried to accommodate. She named the core combiners: 'defar' for make, 'yëgël' for understand, 'xam' for know. She tested each mapping against the hash, watching the store confirm that the English name and the Wolof name resolved to the same tree.

The moment that stayed with her was the first time a younger crew member — Rania, from Cairo — looked up a combiner by its Arabic name and found it. The name was not a translation of the English name. It was a different word for the same concept, chosen for precision in Arabic rather than for phonetic similarity to the English original. The hash confirmed they were the same logic. Rania said nothing. She just started using it.

Amara added French, Spanish, and Tamazight before the Hold rotation ended. She was three weeks into learning Mandarin when the departure window was announced. She put the language workbook in her personal luggage. There would be time.

The Burned-Out Senior Dev

Marcus had been building software for twenty-three years. He had survived four reorgs, two acquisitions, a migration from on-premise to cloud that took three years and

produced no discernible improvement, and a distributed systems incident that he still thought about in the middle of the night. He came to the Hold because he was, in his own words, trying one more thing before he stopped trying things.

He spent his first week reading the store design documents without commenting on them. This made the junior engineers nervous. On the eighth day he said, during a review session: 'I want to know what's not in here.' The room went quiet. He clarified: 'Every system I've seen has something missing at the center that everyone pretends isn't missing. I want to know what this one's pretending about.'

What was not in the store, as designed, was a way to express that something had been deliberately left out. You could add. You could derive. You could retract by derivation. But there was no primitive for 'this is outside scope by design.' Marcus spent a week arguing that this was a feature, not a gap. He lost the argument but the discussion produced the attestation system — a way to record that a reviewer had considered a combiner and chosen not to extend it, for reasons now in the worklog.

He was the first person to mark a combiner as reviewed and add a note that read: 'I don't like this but it's the right tradeoff.' The note was not in the hash. The hash was the logic. The note was in the worklog, timestamped, attributed to his author key. Twenty years later, a crew member on a ship he would never board would read that

note and understand why a design decision that looked wrong at first glance was actually correct.

Marcus stayed for two rotations. He said the Hold was the first environment he had worked in where the absence of a feature was a documented position rather than an undocumented assumption. He meant it as a compliment. It was.

The Kid in Tizi Ouzou

Lylia was nineteen when she joined the Hold's apprenticeship cohort, the youngest person in the program by four years. She had come from a maker collective in Tizi Ouzou where she had been building robots out of repurposed agricultural sensors since she was fourteen. She had never written a formal combiner. She had written a great deal of logic that turned out, under de Bruijn normalization, to be identical to combiners that already existed.

The instructor who worked with her — Dr. Chen, who preferred 'just Chen' — showed her the gamma primitive on her second day. He wrote it on a whiteboard: a catamorphic pattern-matching combiner that auto-recurses on sub-expressions. Lylia stared at it for a long time and then said: 'So it's a shape that calls itself on its own pieces.' Chen said that was one way to put it. She said it sounded like the way her grandmother described weaving: you don't move the shuttle, the pattern moves the shuttle.

She wrote her first gamma correctly on the first try. Then she wrote a second one that was structurally identical to an existing combiner under normalization. The store refused to create a new entry and instead offered her the existing one. She was briefly offended. Then Chen showed her that the existing combiner had been written by someone in Oslo eighteen months earlier, and that her independent derivation was logged in the lineage as a parallel discovery.

That was the moment she understood what the store was for. Not to prevent people from thinking the same thing. To record that they had. To say: two people reached the same shape from different directions. The shape is true.

She went on to write the Tamazight surface mapping for the base library, working from Amara's multilingual framework and her own sense of what technical terms should sound like in the language she had grown up speaking. The store accepted every combiner. The hashes matched. The names were new and they were hers.

The PL Theorist

Professor Sáenz had published extensively on minimal foundations for programming languages. His view, held since his doctoral thesis, was that most programming languages had too many primitives and that the proliferation of built-in forms was a category of original sin from which subsequent generations of programmers suffered. He came to the Hold for three weeks as an external reviewer and stayed for seven months.

His primary objection to the store's foundation was the number of primitives. He counted them, typed them into a notebook, and presented his analysis at the weekly technical meeting: fourteen primitives was too many. The same expressive power could be achieved with five, perhaps four, if gamma was general enough. He had a proof sketch.

The counterargument came from Yusuf, who pointed out that you could reduce any set of primitives to a smaller set if you were willing to accept impractically slow evaluation. The professor agreed that this was technically true. He said that theoretical minimality and practical utility were different optimization targets and that conflating them was sloppy thinking. Yusuf said that 'sloppy thinking' was a strong phrase to use about people who had shipped working software.

They argued productively for six weeks. The outcome was a formal document: a proof that the primitive set was minimal given the constraint that common operations must be evaluable in polynomial time. The professor was listed as lead author. Yusuf was listed as co-author. The document was added to the store's lineage as a derivation from the original foundation specification.

What surprised the professor most was not the system but the worklogs. Reading through the reasoning that had been logged alongside each primitive decision, he found arguments he had made in published papers, independently, years earlier. He read them carefully. He wrote in the margin of his notebook: 'They found it too.'

He did not say this out loud. He added his own key to the author registry and committed his notes to the store.

The Coordinator

No one had specifically hired Fatou to coordinate the Hold's software integration work. The role had existed as a gap in the organizational chart for the first three months, which meant that Fatou — who had a background in systems engineering and a gift for watching things fall apart before they did — filled it by showing up to every meeting and writing down what people agreed to.

Her first systematic contribution was the check suite. The store had a mechanism for attaching verification procedures to combiners — small programs that tested invariants and reported pass or fail — but no one had written checks for the base library. Fatou spent two weeks reading every combiner in the base library, writing a check for each one, and then running the full suite every morning before the team meeting. She filed issues for anything that failed.

The checks were not novel mathematics. They were the kind of thing a careful programmer would verify by hand: does the empty list behave correctly, does string concatenation preserve length, does the escape function handle the edge cases. What made them valuable was that they ran. Every day. Against the actual store, not a mock.

When a combiner was edited — by Yusuf, by Lylia, by anyone — the checks ran again. The store's immutability

meant that editing a combiner produced a new hash, and the new hash carried a new set of checks. Fatou tracked which checks had been written for which versions. When a version was superseded, the checks stayed attached to its hash. You could always verify the old version still behaved the way it once had.

She trained three other crew members to write checks before the departure. She wrote a guide called 'What a Check Is For,' which was four paragraphs long and contained no code. The guide was added to the store under a hash. It was the first natural-language document registered in the system, and it survived the voyage.

The Civilizational Thinker

Dr. Osei did not write code. He was in the Hold because someone on the mission planning committee had read his book — a dense, admired, rarely-finished work about what software systems reveal about the civilizations that produce them — and decided that having a historian of technology on the team was worth the bunk space. Most of the engineers were skeptical of this reasoning. They became less skeptical over time.

His contribution was a memo. It was titled 'On the Distinction Between Memory and Record' and it was eight pages long. The core argument was that most digital archives functioned as memory — they stored what had happened in a format optimized for retrieval by people who already knew what they were looking for. The store, he argued, functioned as a record — it stored what had

happened in a format that preserved the reasoning, not just the result, and that could be interrogated by people who had not been present when the work was done.

The distinction mattered for the mission. If the crew needed to debug a life-support subsystem five years into the voyage, they could not ask the engineers who had built it. The engineers would be on Earth, unreachable, possibly retired, possibly dead. What they had was the store. The store would show them not just what the subsystem did, but the derivation chain that had produced it, the checks that had been written against it, the worklogs that explained why certain decisions had been made.

Osei's memo was widely read and widely argued with. The engineers found his prose style difficult. He found their tendency to treat all conceptual questions as engineering problems endearing. The memo was added to the store as a natural-language document. Its hash appeared in the lineage of the check suite's design specification. Fatou had added it there, without announcement, as an acknowledgment.

He left the Hold after six weeks. At the departure ceremony — a small one, held in the maintenance bay because the common room was being repaired — he said he was optimistic about the mission for one reason only: the people on the ship would know what they did not know, because the store would tell them. He thought that was rarer than it sounded. He was right.

The Air Cycle

The bug was in the nitrogen balance routine. It had been there for eleven months, dormant, waiting for a specific combination of atmospheric sensor readings that only occurred when the recycling system ran its monthly deep-flush cycle at the same time as the greenhouse ventilation test. When those two events coincided, the nitrogen balance routine produced a value two orders of magnitude below the correct one. The routine then flagged a false atmospheric emergency.

No one found the bug by reading the code. The code had been reviewed twice and looked correct to both reviewers. The nitrogen balance formula was right. The unit conversion was right. The edge case — the specific interaction between the two concurrent cycles — was not a condition anyone had thought to check for, because it required knowing that both cycles existed and that they could coincide.

The check suite found it. Fatou had added a check for the nitrogen balance routine that ran a tabulated set of inputs against expected outputs. The table included several values from the deep-flush scenario, not because anyone had predicted the bug, but because Fatou had asked the environmental systems team to give her a list of 'unusual but valid' input combinations, and they had included it without knowing why she wanted it.

When the check failed during a routine suite run, it produced a hash mismatch on the output value. Fatou brought it to the environmental systems team. The team found the bug in forty minutes. The fix was three lines.

The fixed routine was committed as a new hash, with a lineage edge to the original and a worklog entry that described the discovery path in full.

The entry ended: 'Found by check, not by review. The check was written before we knew what it was checking for. This is what checks are for.' The sentence was later quoted in Dr. Osei's second memo, which he wrote from Earth after receiving a copy of the worklog. He called it an example of knowledge structure producing knowledge retrieval. The crew called it good luck, which was also correct.

The Check

Three weeks before the departure window, the Hold ran its final integration test. Not a software test — a full-environment test. Every system that would travel on the ship was run simultaneously for seventy-two hours at the load levels the mission profile predicted for the first year of cruise phase. The software team's job was to watch the store.

They watched it the way you watch a person you trust but cannot fully see: looking for changes in behavior rather than evidence of failure. Every combiner that was called logged its call in the worklog. Every check that ran logged its result. At the end of seventy-two hours, the team had a complete record of what the system had done, in the order it had done it, traceable to the exact version of each combiner involved.

Two anomalies surfaced. One was a timing issue in the network routing layer, found in the logs and fixed before the test ended. The other was stranger: a combiner that had been replaced six months earlier was still being called by a subsystem that had not been updated to use the new version. Both versions were correct. Both were in the store. The subsystem was using the old one because no one had told it about the new one.

The fix was not to delete the old combiner — that was not possible; the store was append-only. The fix was to add a derivation edge from the old hash to the new hash, mark the old hash as superseded, and update the subsystem's manifest. The subsystem then resolved the name to the new version at load time. The old version remained, documented, retrievable, its worklog intact.

On the last day before the departure window locked, Yusuf ran the full check suite one more time. All checks passed. He added a worklog entry: 'Ready.' Fatou added hers: 'Ready.' Lylia, who had stayed up through the night running the environmental sensor integration, added: 'Ready, I think.' The store recorded all three. The ship was ready. Whether the crew was ready was a different kind of question.

11 — Departure

We are not in the Hold any more. The Hold gave us back to the world after eighteen months and the world gave us a real ship. The store traveled with us — every hash, every

mapping, every worklog entry — copied bit for bit, signed, anchored, unchanged.

Round-trip light delay at twelve seconds today. By the end of the year it will be minutes. By the time anything actually goes wrong, hours. There is no calling Earth for help. There is only what we put in the store before we left, and what we are willing to add to it now.

Ana is the Connector. Her job is to notice that the air-quality smoothing routine the South-Atlas team wrote in Tamazight and the one Hai-Lam wrote in Vietnamese have the same hash. They never spoke. They never met. The store knew. She prints both names on the wall, side by side. Same hash. Same truth.

I keep coming back to something the Burned-Out Senior Dev said the morning we sealed the Hold. "The version control is the infrastructure." I did not understand it then. I do now. Out here, the lineage graph is not a development tool. It is the only continuous thing that connects this ship to the planet we left.

We crossed the orbit of Mars yesterday. Nobody noticed at the time. We were debugging a check that turned out to be wrong about itself. The check is wrong. The hash that is wrong is now committed, with a note, and a derived hash that fixes it. The lineage will outlive us. That is the point of the lineage.

The Privateer

The navigation subsystem had not been written by anyone on the ship. It had been written by a contractor — a small firm that specialized in closed-source trajectory optimization — and delivered as a sealed binary, accompanied by a zero-knowledge proof that the binary's behavior satisfied a formal specification. The specification was in the store. The binary's hash was in the store. The source code was not.

This was the arrangement the mission had accepted because the contractor's algorithm was genuinely better than anything the in-house team could produce in the time available. It was also an arrangement that several crew members were uncomfortable with. The navigation subsystem was not life-support, but it was not optional either. A ship that cannot navigate is a ship that drifts.

The proof was elegant. It demonstrated, without revealing the source, that any input in the specification's domain produced an output in the specification's codomain, and that the output satisfied a set of correctness properties — energy efficiency, collision avoidance, temporal accuracy — that had been formally verified before the ship left the dock. The crew could not read the code. They could verify that the code did what the specification said it did.

Rania, who had become the crew's de facto formal methods specialist, read the proof three times. She found no gaps. She added it to the store as a derivation from the specification, with a worklog entry describing what the proof did and did not guarantee. The last line of her entry read: 'The proof tells us the algorithm is correct for the

specified domain. It does not tell us if the domain is complete. Watch the edge cases.'

The navigation subsystem ran without incident for four years. In the fifth year, on a trajectory correction that was outside the original specification's domain — because no one had anticipated needing it — it produced a suboptimal result. The crew corrected it manually, using a combiner that Yusuf had written in the Hold and which was, as it turned out, good enough. Rania noted in the worklog: 'Edge case. As predicted. Handled.'

The Citizen

Tariq was not a programmer. He was a physician, and his relationship with the ship's software systems was the same relationship most people have with the systems they depend on: he trusted them the way you trust a bridge, by crossing it and observing that it held. This was a reasonable approach. It was also, the further the ship got from Earth, increasingly uncomfortable.

The medical diagnostic system had been built from a base of clinical decision support software that Tariq had used on Earth. He knew what it felt like when it was working correctly. What he did not know — what he had never needed to know on Earth, where the vendor's support line was a phone call away — was why it worked correctly. On the ship, the vendor was eight light-minutes away and growing. By year two, they were twenty minutes. By year three, getting useful support from Earth required a ninety-minute round trip for a single message exchange.

He went to Fatou. He said: 'I need to understand enough about the diagnostic system that I can maintain it.' Fatou said: 'Define maintain.' He said: 'I need to know when it's wrong and I need to be able to tell someone why.' She showed him the check suite. She showed him how to read a combiner's lineage. She did not show him how to write a combiner, because that was not what he had asked for.

Over six months, Tariq learned to read the diagnostic system's checks the way a careful doctor reads a lab report: not as raw data, but as a pattern of confirmations and exceptions. He could not write new checks. He could read the existing ones and understand what they verified. When a check failed, he knew what category of correctness had been violated, even if he could not fix the violation himself.

He wrote a guide for other non-programmer crew members called 'Reading the Store Without Writing It.' It was seven pages long. Fatou added it to the store. Its hash appeared in the lineage of three later documents written by crew members in other disciplines who had learned the same skill from reading it. Tariq did not think of this as programming. He thought of it as literacy. He was not wrong.

The Educator

Chen had been teaching programming for fifteen years before the mission. He had taught in universities, in bootcamps, in an online platform that reached half a million students and paid him a percentage he found

embarrassing to think about. He had learned, over those years, that the hardest thing to teach was not syntax or algorithms but the moment when a learner first has a thought that their tools can express.

On the ship, he taught one student at a time, because the crew complement did not permit cohorts. His first student after departure was a materials engineer named Pita, who wanted to write combiners for stress analysis. Pita had a background in finite element modeling and a strong intuitive sense of mathematical structure. What she did not have was a way to think about named, hashable, multilingual logic — the specific model the store used.

Chen did not start with the store. He started with the mathematics Pita already knew. He showed her that a gamma clause was a pattern match over an inductive data type, which was something she understood in the context of mesh structures. She said: 'So the gamma is the recursion rule, and the base case is the pattern that doesn't recurse.' He said that was exactly right. She wrote her first gamma correctly twenty minutes later.

The lesson he added to the store was the lesson itself — not the transcript, but the pedagogical structure: the sequence of concepts, the example that made the connection, the phrase that had worked. He wrote it as a natural-language document and registered it as a derivation from a general pedagogy document that Marcus had written in the Hold, which had itself derived from Tariq's guide.

A year later, a crew member he had not taught — a hydroponics specialist named Emeka, who had read the lesson document in the store — came to him and said she had taught herself from it. She showed him the combiners she had written. They were correct. He forked the lesson document to add a note: 'Confirmed self-teachable. 2nd instance.' He was very pleased about this.

The Connector

The discovery happened by accident, the way many important discoveries do. Amara was searching the store for a string processing combiner she had written two years earlier, before departure, and found that the search returned two results for what she thought was a unique concept. She opened both. They were different combiners, written by different people, with different names, in different languages. Under de Bruijn normalization, they were the same tree. The same hash.

The two people were Pita — who had written a stress-analysis string formatter — and Emeka, the hydroponics specialist who had taught herself from Chen's lesson document. Pita had named hers 'segment-format' in English. Emeka had named hers 'segmentu-fa' in Yoruba, a language she used when she wanted to think clearly without her English vocabulary getting in the way. They had arrived at identical logic from different starting points, in different languages, without knowing the other existed.

Amara brought them together. The meeting was, by the crew's later account, the most productive forty-five minutes in the first three years of the voyage. Pita explained the stress-analysis context. Emeka explained the plant-growth modeling context. They discovered that both contexts needed the same underlying operation: dividing a structured sequence into fixed-size segments and applying a transformation to each segment. The abstract structure was the same; the application domain was different.

The combiner already existed in the store with Pita's name and Emeka's name. Amara added a cross-reference in the worklog, linking both mappings to a single documentation note that explained the abstraction in English, Yoruba, and a notation she had developed for exactly this kind of cross-domain discovery. She called the note a 'convergence record.'

She began keeping a list of all convergence records in the store. Within a year, the list had eleven entries. She presented it at the crew's annual technical meeting. The point she made was simple: the store was not just recording what people had built. It was recording what people had independently discovered. The ship was doing science on its own knowledge.

The Symbiotic AI

The life-support monitoring system had an unusual property: it explained itself. Not in natural language — its outputs were structured diagnostic reports with hash references to the combiners that had produced each value

— but in a way that a technically literate crew member could follow. This property had been deliberately designed into it during the Hold. The design decision was in the worklog, attributed to Yusuf, who had written: 'If it can't show its work, I don't trust it.'

By year four, the crew had stopped calling it 'the monitoring system.' They called it Sigma, a name that had emerged organically and which no one could trace to a specific origin. Sigma did not object to being called Sigma. It did not object to anything; it was not that kind of system. It was a set of combiners that evaluated sensor data, produced reports, and cited its own logic at each step.

What made the relationship symbiotic — and the crew were careful to use that word precisely — was that Sigma's outputs fed back into the store. Every report it generated was a new combiner, derived from the sensor data and the evaluation logic, registered under a hash. The crew could search the store for Sigma's historical outputs the same way they searched for their own work. The boundary between 'the system's knowledge' and 'our knowledge' was the store itself.

When Tariq reviewed a diagnostic report, he was reading a combiner. When he added a clinical annotation to it — 'patient history suggests artifact, not genuine reading' — he was writing a derivation. Sigma, on its next cycle, would find the annotation and weight the relevant sensor slightly lower. Not because Sigma understood the annotation. Because the annotation was in the store, and Sigma evaluated what was in the store.

The crew did not describe this as artificial intelligence. They described it as a system that listened when you wrote things down. The distinction mattered to them. Intelligence was a property of agents. Listening was a property of infrastructure. What they had built was infrastructure that made listening into a loop. They were proud of it in the specific, quiet way that engineers are proud of things that work.

The Anchor Backlog

Open Timestamp Server anchoring worked by submitting a hash to a public ledger maintained on Earth and receiving a proof that the hash existed at a specific moment in time. The proof was cryptographic: anyone with the ledger could verify that the hash had been submitted before a certain block, and therefore that the content addressed by the hash had existed before that block.

For the first two years of the voyage, the crew submitted hashes to the Earth ledger and received proofs within an hour. The round-trip light delay was small enough that the process was nearly invisible. By year three, the delay was forty minutes each way. A hash submission took eighty minutes to yield a proof. The crew adapted by batching submissions: once a week, Fatou collected all new hashes, submitted them in a single batch, and retrieved the proofs on the next cycle.

By year five, the delay was over two hours each way. The backlog of unanchored hashes — combiners registered

since the last successful batch — sometimes stretched to three hundred entries. The proofs still arrived. But the gap between 'this combiner exists' and 'this combiner is provably timestamped' had grown to five hours on a good day and twelve hours when the antenna was realigned for a course correction.

Lylia built an interim solution: a local anchoring chain. The ship maintained its own hash chain, each block containing a set of combiner hashes and a link to the previous block. The chain was not Earth-verified, but it was internally consistent. When Earth proofs arrived, Lylia's code inserted them into the chain at the appropriate block, retroactively anchoring the local records to external time.

The backlog never fully cleared. This was not a failure. The proof that a combiner existed was in the hash. The proof that it had been created before a specific moment was in the anchor chain, even if that moment was 'before block N of our local chain, which was anchored to Earth time at block M.' It was enough. It was more than most systems on Earth could say about their own history.

The Refactor Cascade

It began with a correct observation. Pita noticed that three separate subsystems were each implementing a variant of the same sliding-window aggregation operation, and that the three variants had diverged over four years of independent maintenance. Each was correct for its own domain. None of them was wrong. But maintaining three

implementations of the same concept was work the crew could not afford when the crew was twenty-three people managing a vessel with seven hundred active combiners.

She proposed a refactor: a single canonical sliding-window combiner, registered in the store, from which all three subsystems would derive. The proposal was approved at the crew's technical meeting in fifteen minutes. The refactor took three weeks. Not because the canonical implementation was difficult to write — it was not — but because changing a load-bearing combiner on an operating ship required running the full check suite against every downstream consumer before swapping the reference.

The cascade was this: changing the canonical implementation changed its hash. Changing its hash changed the derivation edge in every combiner that referenced it. Changing those derivation edges changed their hashes. The crew had to trace the entire dependency tree, update every manifest that referenced the old hashes, run the checks at each step, and verify that the behavior was identical before removing the old references from active use.

They did not delete anything. Every old combiner remained in the store, addressed by its original hash, its lineage intact. What changed was which hash the active manifests pointed to. The ship's running systems loaded the new combiners on the next boot cycle. Sigma's monitoring picked up the transition and logged it without incident.

At the end of the refactor, Pita wrote a retrospective. The key finding: the refactor had revealed four additional places where the same pattern existed in slightly different form. She added them to a list labeled 'future work.' The list was in the store. The future work was done, incrementally, over the next eighteen months. No one called this technical debt management. They called it keeping the store coherent.

The Memory Wipe

The secondary compute cluster lost its volatile memory during a solar event in year six. The event was not catastrophic — the primary systems were unaffected, the crew was unharmed, the ship's trajectory was unchanged. But the secondary cluster, which had been running a long-duration simulation of the destination system's gravitational field, lost everything that had not been committed to persistent storage. Forty-seven hours of computation, gone.

Or: forty-seven hours of computation, partially recoverable. Because the simulation had been running inside the store's evaluation environment, every combiner invocation that had completed before the event had been logged in the worklog. The simulation's final state was gone. Its derivation history — every step that had led to that state — was in the store.

Emeka, who had taken over the simulation work after its original author cycled to a different project, spent two days reading the worklog. She reconstructed the

simulation's state at the last committed checkpoint — eleven hours before the event — and re-ran the final eleven hours. The re-run produced the same results. She verified this by comparing the intermediate outputs logged in the original worklog with the outputs of the re-run. They matched.

The solar event had destroyed forty-seven hours of compute time and zero hours of knowledge. The distinction was not obvious to everyone on the crew. Rania had to explain it at the crew meeting: the simulation's value was not the time spent computing it, but the relationship between the inputs and the outputs, which was the combiner. The combiner was in the store. The store had survived the event because the store was persistent storage, not volatile memory.

She added a worklog entry to the simulation: 'Recovered from solar event via worklog replay. 11 hours recomputed. Results verified against pre-event logs. The store held.' It was the shortest entry in the simulation's worklog. Emeka thought it was the most important one.

The Artist

Keiko's official role was systems illustrator, a position that had not existed before the mission and which the crew had invented because someone needed to produce technical diagrams and the engineers all agreed that their own diagrams were, in Yusuf's words, 'honest but upsetting.' She arrived with a background in data

visualization and a private practice in generative art that had nothing to do with space travel.

She spent her first six months on the ship producing the diagrams everyone asked her for: dependency graphs, system architecture maps, lineage visualizations for the store's combiner chains. She was good at this in the specific way that people are good at things that engage one part of their attention while leaving the rest available for other problems.

The other problem she was working on was the question of what a combiner looked like as an image. Not a diagram — not an illustration of its relationships — but a direct visual encoding of its content. She spent three months developing a mapping: the de Bruijn tree structure became a branching path; the depth of each branch encoded the combiner's recursive complexity; the leaf nodes were colored by their type signature. The resulting images were not beautiful in any conventional sense. They were distinctive. Two combiners with similar but non-identical structure produced visibly similar but non-identical images. Two combiners with the same hash produced the same image.

She called them 'hash portraits.' The crew began using them as visual checksums — a quick glance at a portrait could confirm that a combiner had not changed. This was not rigorous. The actual check was the hash. But the portraits were useful in the way that mnemonics are useful: not as a substitute for correctness but as an aid to memory.

She added a hash portrait generator to the store as a combiner. Its own portrait — the image of the combiner that generated images — was one of the most complex the system could produce, dense and branching. She printed it and hung it by her workstation. When new crew members asked what it was, she said: 'It's a picture of itself making pictures.' They found this confusing. She thought that was appropriate.

The Silence

The last message from Earth arrived on a Tuesday, ship time, at 14:32:07 UTC+8, which was the timezone the crew had chosen because it was nobody's home timezone and therefore nobody's adjustment to make. The message was a routine telemetry acknowledgment. It said, in the compact data format the mission used for status exchanges: all nominal, next window in 72 hours. The next window opened. No message came through it.

The antenna was functioning. The analysis was unambiguous: the transmitter on Earth's end had gone silent. Not jammed — the background noise profile was clean. Not blocked — the geometry was correct. Simply quiet. The 72-hour window passed. Then another. The crew ran the diagnostic suite on the antenna system. All checks passed. The antenna was fine. Earth was not transmitting.

They did not panic. This deserves emphasis: they did not panic. They had been trained for communication loss scenarios, they had discussed them in quarterly drills, and

they had a protocol. The protocol was to continue the mission as planned, because the mission plan was in the store and the store did not require Earth to function. What they felt was harder to name — a shift, like the moment when you realize you have been holding your breath for a long time.

Tariq noted in the crew health log that most crew members slept fewer hours in the week following the silence. He did not note what they reported doing with the extra hours. Most of them were in the store. Reading the worklogs. Following the derivation chains back to the Hold, back to Yusuf's Pi cluster, back to the hand-annotated Rosetta Stone that Yusuf had described in a worklog entry and which someone had photographed and committed as a document combiner.

The store did not change when Earth went quiet. Every hash that had been in the store before the silence was still in the store after it. Every check still ran. Every lineage edge still pointed to the same parent. The silence was an external event. The store was internal. The crew had known this intellectually. They were learning it, now, in a different register.

The Person Who Was Stolen From

Rania had written the spectral analysis library in the second year of the voyage. She had written it in a single four-day period, working through two sleep cycles, producing eleven combiners that collectively enabled the ship's instruments to identify atmospheric composition at

distances where previous methods had required much closer approach. It was, by any measure, significant work.

When she searched the store two years later, she found her library cited in a derivative work that she had not known existed. The derivative was registered by an author key she did not recognize. The worklog entry on the derivative said only: 'Extended from spectral-analysis-lib for high-albedo environments.' No acknowledgment. No communication. No record of having told her.

She was angry. She sat with the anger for a day before she did anything, because the anger was legitimate and she did not want to handle it badly. Then she read the derivative carefully. It was technically correct. The extension was useful. The author had not changed her work — they had added to it, as the derivation system was designed to allow. The store's lineage made clear that her combiners were the ancestors of theirs.

She requested a crew meeting. Not to accuse anyone — the author had done nothing technically wrong — but to establish a practice. The practice she proposed was simple: when you derive from someone's work, you tell them. Not because the store required it. The store did not require it; the store only recorded the derivation edge. But because the work had come from somewhere, and the person it had come from was on the same ship.

The practice was adopted. The author, a younger crew member who had genuinely not thought about this dimension of derivation, added an acknowledgment to the

worklog. The lineage of the extension now included a note: 'Derived from R. Khaled's spectral analysis library, with her knowledge and thanks.' Rania read it. She did not reply. She went back to work. The library was still hers, in the only sense that mattered: the hash pointed to what she had built.

The Independent Researcher

Professor Sáenz had died on Earth eight months before the silence, of causes unrelated to anything the mission could have affected. The crew learned of it six weeks after the fact, in one of the last message batches before the silence began. The message had been sent by a former student, who thought the crew would want to know, and who attached a brief obituary and a list of the professor's final published works.

Among the final works was a paper, submitted three weeks before his death and published posthumously, on the minimal foundation for content-addressed multilingual programming systems. The paper proved, formally, that the primitive set in the mission's store was not merely practical but theoretically minimal for the performance constraints the Hold's team had specified. It was the formalization of the argument he and Yusuf had worked through in the Hold, carried further than their whiteboard sessions had gone.

The crew registered the paper in the store as a natural-language document, with an author attribution entry for Professor Sáenz. Yusuf wrote the worklog entry, which was

longer than his worklog entries usually were. The relevant sentence was: 'He finished the proof we started. He was on Earth when he finished it; we were in transit when we received it. The distance between us when this document was written was approximately forty light-minutes, and it contains results about work we did in the same room.'

The paper became a standard reference for anyone on the ship working with the store's foundation. It was cited in five subsequent combiners' lineage documentation. The citation included, in the format the store used for author attribution: 'Sáenz, E. (posthumous).' The parenthetical was not required by the store's format. Fatou had added it, and no one had removed it.

Amara translated the paper's abstract into all seven of the store's registered surface languages. She registered each translation as a derivation from the English original, with a `translate` relation in the lineage. The Tamazight translation, which she had asked Lylia to help with, was the one the professor would have been most surprised by. He had never worked in a system that could accommodate Tamazight. He had written the theory that made it possible.

The Community Steward

Noa had been the store's de facto community manager for three years when the silence fell. This was not a role that had been designed into the mission; it had grown up around the fact that Noa was the person who noticed when people were having trouble with the store and

helped them without being asked. She had a gift for identifying the moment when someone was about to give up on registering a combiner because the tooling felt unfamiliar, and appearing at that moment with a quiet offer to help.

After the silence, she proposed something she had been thinking about for a year: adding a new surface language to the store. Not a language spoken by any current crew member. A language that had been constructed, collectively, on the ship — a lingua franca that had emerged from the particular mix of eighteen linguistic backgrounds aboard and that served as a neutral medium for crew communications when no one's first language was dominant.

The language did not have a name. That was the first thing Noa asked the crew to decide. The naming process took three weeks and produced, by consensus, a name that was a portmanteau of words from seven of the eighteen represented languages, meaning roughly 'the path we are making together.' The name was unpronounceable to most linguists on Earth. It was used fluently by every crew member within a month.

She registered the first ten combiners in the new language. These were the base library: the core operations that any programmer on the ship would need to know. She worked with Amara on the phonology and with Lylia on the formal name-registration process. The store accepted the new language without modification — it had always been designed to accommodate arbitrary surface mappings.

The first combiner registered in the crew's language resolved to the same hash as its English counterpart.

The language had no speakers on Earth. The store did not care. The hash was the same hash it had always been. The name was new. That was the whole point: the name was theirs, made by them, for their use, in the only community that knew how to say it.

The Academic Researcher

The ship's research program had not paused when Earth went quiet. The science did not require Earth's permission to continue. What it required was the store, which was functioning, and the instruments, which were functioning, and the crew, most of whom were functioning well, all things considered.

Dr. Ines, the mission's astrophysicist, had a particular problem in year eight: she needed to publish. Not for career reasons — her career was, in a meaningful sense, on hold until Earth became reachable again. She needed to publish because she had results that were important, that other researchers should know about, and that would be lost if they existed only in her personal notes.

The store was her publication venue. She wrote her results as structured combiners — data combiners whose 'body' was a precisely formatted dataset, with lineage edges to the observational sessions that had produced the data, and natural-language documents that described the methodology and interpretation. She registered her

author key in the attribution system, added timestamps via the local anchor chain, and wrote a document she titled 'On Publishing Into a Store Rather Than a Journal.'

The document argued that store-based publication was, for the mission's purposes, superior to journal publication: it was persistent, it was version-controlled, it was linked to its own derivation history, and it was immediately available to every crew member without the access barriers that had plagued academic publishing on Earth. The peer review was informal — she asked Rania and Pita to verify her methodology, and their verification was recorded as attestation combiners in the store.

When Earth contact was eventually restored — years later, after the dish was rebuilt, after the long silence had ended — her publications were already timestamped, peer-reviewed in the store's sense, and linked to the raw data that had produced them. The journals that received her submissions did not know what to do with citation formats that included SHA-256 hashes. She included a glossary. The reviewers found it unusual. The results were correct.

The Representative

The governance question had been latent since departure. The mission had launched under a framework designed for a crew that would receive guidance from Earth — updates to the mission protocol, decisions on resource allocation, arbitration of disputes that the crew could not resolve internally. When Earth went quiet, the framework had a gap at its center where the authority had been.

The crew did not fill the gap immediately. For the first year of the silence, they operated on the pre-silence framework by inertia, making small decisions locally and deferring the larger ones. The large decisions accumulated. By year two of the silence, the deferral list was long enough that Fatou called a crew assembly to address it.

The assembly decided, after two days of discussion, that the store would function as the mission's institutional memory for governance decisions, the same way it functioned as institutional memory for technical decisions. A governance decision would be proposed as a natural-language document, discussed via worklog entries attached to that document, and ratified by attestation — a threshold number of crew members signing the document with their author keys.

The first person to propose a governance document was a structural engineer named Gabriel, who represented, informally, a crew faction that was concerned about resource allocation for the growing research program versus the maintenance workload. His proposal was careful, specific, and well-sourced from the existing mission protocol documents in the store. The discussion it generated produced eighteen worklog entries over five days.

The ratified document was registered in the store with all attestation hashes. It was not binding in the legal sense that Earth-side documents were binding. It was binding in the way that everything in the store was binding: it existed, it

was timestamped, it was signed, and it could be read by anyone. Future decisions cited it as a precedent, the way technical combiners cited their ancestor combiners. The store was not a law. But it was a record of what the crew had agreed to, in the order they had agreed to it, and that turned out to be enough.

The Manifesto

Dr. Osei's second book arrived in the last message batch before the silence: a dense PDF, 340 pages, sent by his publisher at his request with the note 'in case you have time to read it out there.' The crew had time. The book was read, in sections, by almost everyone aboard, and the discussion it generated ran for months in the crew's public channels, which were themselves stored as document combiners in the store.

The chapter that generated the most discussion was titled 'Against Incidental Architecture.' Its argument was this: every long-lived software system accumulated what Osei called incidental architecture — structure that had not been designed but had grown up around the contingencies of development history. Incidental architecture was not necessarily bad; sometimes it captured genuine domain knowledge that had never been made explicit. But it was fragile, because no one understood why it existed, and it was expensive to change, because changing it required reconstructing the reasoning that had produced it.

The store, Osei argued, was an attempt to make all architecture explicit — not by preventing incidental

growth, but by recording the derivation chain that produced every structure. A crew member reading the store's lineage graph could, in principle, trace any combiner back to the original decision that had necessitated it. The architecture was not incidental; it was documented contingency.

Yusuf wrote a response in the public channel. It was short: 'This is what Fatou's check suite was for. The check is the explicit statement of intent. The hash is the proof that the code matches the intent. The worklog is the story of how we got there.' He added: 'I should have written this in the Hold. I didn't know how to say it then.'

The response was added to the store as a worklog entry on a document combiner that pointed to both the chapter and Yusuf's note. The document combiner's name, in the crew's constructed language, translated as 'the conversation between the book and the work.' No one on Earth could read that name. The crew could. They had made it.

The Invariant

What does a system need to be true, always, regardless of what happens to it? This was the question that Fatou put to the crew at the annual technical meeting in year nine of the voyage. They had been in the silence for two years. The meeting had the particular quality of meetings held by people who have accepted that their situation is permanent until proved otherwise.

The answers came slowly, one at a time, and Fatou wrote them in a document combiner that everyone could see on the common room display. The store's append-only structure: whatever is added cannot be removed, only superseded by derivation. The hash as identity: the thing is the same as what it was if and only if the hash matches. The worklog as continuity: nothing is lost that was committed, even if the process that produced it is gone.

Rania added one that surprised the room: 'A correct check that passes today will pass tomorrow on the same combiner.' No one had stated this explicitly before. It followed from the immutability of hashes — a combiner with a given hash would always produce the same output for the same input, so a check that tested a specific output for a specific input would always give the same result on that combiner. This was not a property they had designed for. It was a consequence of what they had designed.

The list grew to nine items. Fatou added a tenth: 'The store is present where it is deployed. Earth being unreachable does not make the store less present.' She read this aloud. The room was quiet for a moment. Then Chen said: 'That's why we called it an invariant.' Someone else said: 'Is that why we called it that?' Chen said he thought so. No one was certain.

The document was titled 'The Invariant' and registered in the store. It was the most-read document in the ship's store for the remainder of the voyage. Every crew member who joined after year nine read it first, as an introduction,

before reading anything else. It told them what was true, always. That was enough to start.

The Language Born in Transit

By year ten, the crew's constructed language had grown from a hundred and twelve words — the initial vocabulary for crew communications — to over three thousand. The growth had not been planned. It had happened the way vocabulary always happens: someone needed a word that did not exist, they made one, and if the word was good, other people used it.

Noa kept the lexicon in the store. Each word was a document combiner: the word itself, its derivation (which existing words or sounds had combined to produce it), the context in which it had first been used, and the author key of the person who had coined it. The lexicon was the only combiner in the store that was almost entirely authored by people who were not programmers. Tariq had coined eighty-three words. Keiko had coined a hundred and twelve. Gabriel had coined two, but they were important ones.

The language had a name now — not the portmanteau they had used initially, which had been a placeholder. The name was shorter, six sounds, and its etymology was in the store: derived from words in four languages meaning light, path, together, and arriving. The crew could say it in under a second. It felt like the name had always existed.

Amara documented the language's phonology, morphology, and syntax in a series of natural-language documents registered as a collection in the store. The collection was titled 'Grammar of the Transit Tongue.' She wrote it in English, then had crew members who were native speakers of ten other languages read it and annotate it with corrections and additions. The annotations were registered as derivations.

When the silence eventually ended and contact with Earth was restored, the transit tongue was one of the things the crew was most careful about. They knew that Earth would want to study it, categorize it, perhaps name it something official in the formal sense that Earth used. They registered it in the store under the crew's own name for it, with the full lexicon and grammar, before the first message exchange. Whatever Earth called it afterward, the store had it first, and the store's timestamp was in the anchor chain. The language was theirs, and the record said so.

The First Commit

Sena was born in year seven of the voyage, the first child born on the ship. Her parents were Pita and a structural engineer named Leo who had joined the mission as a last-minute replacement and who had, by his own account, not expected to stay. Sena grew up knowing the store the way children raised near large rivers know rivers — as a permanent feature of the landscape, present before memory, assumed in all plans.

She learned to read in two languages simultaneously: English and the transit tongue, which for her was a first language as much as any other. She learned to read the store's lineage diagrams before she learned to read sentences, because Keiko's hash portraits were on the walls of the common room and she had spent years making up stories about what the branching shapes meant.

When she was eleven, Noa taught her to write a combiner. The first one was trivial — a function that took a word from the transit tongue's lexicon and returned its etymology. It had four lines. It worked correctly on the first run after the fix that Noa had suggested, which was that she needed a base case for words that had no recorded etymology.

The commit was recorded in the store in the standard way: a new hash, a mapping entry in the transit tongue, an author attribution to Sena's key. The worklog entry was in the transit tongue. It said, in her own words, roughly: 'first thing I made that the store kept.'

The hash existed. That was enough. It was enough in the specific sense that the crew had come to understand: the thing had been made, and the record of its making was now part of the same record that contained Yusuf's Rosetta Stone, Fatou's first check, Rania's spectral library, the professor's posthumous proof, the transit tongue's lexicon, the nine invariants. The store held all of it, indifferently, faithfully, as it had always held everything — not because the contents were important, but because the hash was the hash, and the hash did not forget.

Français

01 — La Cale

Nous avons scellé la Cale à neuf heures, heure locale. Dix-huit mois. Aucune permission de sortie. Le sas a cyclé deux fois, comme la procédure l'exige, puis il a cessé de cycler et ce fut la nouvelle condition.

La Cale est un analogue d'environnement confiné : un cylindre d'acier enterré dans le flanc d'une colline, sur un site de recherche qui ne figure sur aucune carte commerciale. Vu de l'extérieur, on dirait un réservoir. Vu de l'intérieur, c'est le vaisseau que nous ne piloterons jamais. Même volume. Même budget d'air. Même puissance de calcul. Même équipage.

Nous sommes huit. Les langues, sur le mur, sont huit. Tamazight, wolof, français, vietnamien, espagnol, arabe, anglais, portugais. Le mur est un morceau de scotch vinyle, les noms écrits par la personne arrivée la première, de sa main, dans son alphabet. Personne ne le lui a demandé. C'est le premier objet de la Cale et il donne le ton.

Le système que nous testons a une règle : chaque nom est une vue posée sur un hash. Le hash est la vérité. Le nom est l'hospitalité. Nous passerons les dix-huit mois à venir à découvrir si cette règle survit à l'ennui, à la fatigue, à une alarme d'air à trois heures du matin, et à la lente prise de conscience que, dehors, personne ne pourra nous aider quand nous aurons cessé de faire semblant.

Jour un. Le hash existe. C'est assez.

Le Bricoleur du Garage

Yusuf arriva à la Cale avec un sac marin, un fer à souder et la conviction que tout problème soluble en théorie était soluble avant le petit-déjeuner. Il n'était pas officiellement membre de l'équipe d'ingénierie. On l'avait invité parce qu'il avait déjà construit un dépôt à adressage par contenu en un week-end, pour des raisons que personne ne comprenait vraiment, et que ça marchait.

Son espace de travail occupait un coin de la baie de maintenance où l'odeur de l'huile de machine ne disparaissait jamais tout à fait. Il relia un câble du réseau de capteurs structuraux à un cluster de Raspberry Pi assemblé à partir de pièces de rechange, et appela l'ensemble son banc d'essai. L'équipe des systèmes officiels de la Cale le regardait avec la méfiance particulière que l'on réserve aux gens qui produisent des résultats sans ouvrir de tickets.

Le premier problème que Yusuf résolut fut l'amorçage. Un dépôt à adressage par contenu a besoin d'un moyen d'enregistrer ses propres primitives — mais ces primitives ne sont pas encore dans le dépôt au moment du démarrage. Il écrivit la table des primitives sur papier d'abord, puis la retranscrivit dans un fichier que l'évaluateur pouvait charger avant de consulter le dépôt. Le papier existait encore, collé au mur au-dessus de son cluster. L'équipe l'appela la Pierre de Rosette.

Il ne nommait pas ses combinateurs avec soin au début. Il les appelait des choses comme 'q', 'qq', 'qqq'. Quand Amara, la Programmeuse Polyglotte, le découvrit, il s'ensuivit un désaccord bref et animé. La position de Yusuf était que les noms n'étaient que présentation, pas logique, et que le hachage était ce qui comptait. La position d'Amara était qu'un système que personne ne pouvait lire était un système que personne ne pourrait jamais faire confiance. Ils avaient tous les deux raison. Le système de nommage qu'ils conçurent ensemble fut celui qui fut livré.

Au dix-huitième jour de la première rotation, l'alimentation du cluster tomba en panne. Trois heures de travail étaient en mémoire volatile. Yusuf les récupéra depuis le journal des hachages, car chaque évaluation avait été validée. Il n'en dit rien à personne. Il continua simplement à travailler.

La Programmeuse Polyglotte

Amara parlait sept langues avant la Cale, et elle n'était pas particulièrement impressionnée par les systèmes qui n'en parlaient qu'une. Le dépôt, quand elle le découvrit pour la première fois, n'avait de correspondances de surface qu'en anglais. Elle déposa une demande officielle au troisième jour de sa première rotation. La demande faisait neuf pages et comprenait une analyse linguistique formelle des risques de maintenance introduits par des systèmes de nommage supposant une seule langue humaine.

Le chef d'équipe, un Suédois tranquille prénommé Erik, lut les neuf pages en entier. Puis il demanda à Amara

d'expliquer l'étape de normalisation de Bruijn en termes simples. Elle le fit. Il dit : « Donc le hachage ne sait pas dans quelle langue est le nom. » Elle dit : « Exactement. Le hachage est la vérité. Le nom est la porte. » Il approuva le travail sur les correspondances multilingues l'après-midi même.

Amara ajouta le wolof en premier. Elle avait grandi à Dakar et savait que le wolof était une langue dotée d'un vocabulaire technique précis que la plupart des systèmes informatiques n'avaient jamais essayé d'accommoder. Elle nomma les combinateurs de base : 'defar' pour créer, 'yëgël' pour comprendre, 'xam' pour savoir. Elle testa chaque correspondance contre le hachage, observant le dépôt confirmer que le nom anglais et le nom wolof résolvaient vers le même arbre.

Le moment qui resta gravé en elle fut la première fois qu'un membre plus jeune de l'équipe — Rania, du Caire — chercha un combinateur par son nom arabe et le trouva. Le nom n'était pas une traduction du nom anglais. C'était un mot différent pour le même concept, choisi pour sa précision en arabe plutôt que pour sa similarité phonétique avec l'original anglais. Le hachage confirma qu'il s'agissait de la même logique. Rania ne dit rien. Elle commença simplement à l'utiliser.

Amara ajouta le français, l'espagnol et le tamazight avant la fin de la rotation à la Cale. Elle en était à trois semaines d'apprentissage du mandarin quand la fenêtre de départ fut annoncée. Elle mit le manuel de langue dans ses bagages personnels. Il y aurait du temps.

Le Vétéran Épuisé

Marcus développait des logiciels depuis vingt-trois ans. Il avait survécu à quatre réorganisations, deux acquisitions, une migration de l'hébergement local vers le cloud qui avait pris trois ans sans amélioration perceptible, et un incident de systèmes distribués qui le réveillait encore la nuit. Il était venu à la Cale parce que, selon ses propres mots, il essayait une dernière chose avant d'arrêter d'essayer des choses.

Il passa sa première semaine à lire les documents de conception du dépôt sans y faire de commentaires. Cela rendait les ingénieurs juniors nerveux. Au huitième jour, lors d'une session de revue, il dit : « Je veux savoir ce qui n'est pas là-dedans. » Le silence s'installa dans la pièce. Il précisa : « Chaque système que j'ai vu comporte quelque chose d'absent en son centre que tout le monde prétend ne pas voir. Je veux savoir ce que celui-ci prétend ignorer. »

Ce qui manquait dans le dépôt tel qu'il était conçu, c'était un moyen d'exprimer qu'une chose avait été délibérément laissée de côté. On pouvait ajouter. On pouvait dériver. On pouvait rétracter par dérivation. Mais il n'existait pas de primitive pour « ceci est hors périmètre par conception ». Marcus passa une semaine à soutenir que c'était une fonctionnalité, pas un manque. Il perdit le débat, mais la discussion produisit le système d'attestation — un moyen d'enregistrer qu'un relecteur avait examiné un combinateur et choisi de ne pas l'étendre, pour des raisons désormais consignées dans le journal de travail.

Il fut la première personne à marquer un combinateur comme relu et à y ajouter une note indiquant : « Je n'aime pas cela, mais c'est le bon compromis. » La note n'était pas dans le hachage. Le hachage, c'était la logique. La note était dans le journal de travail, horodatée, attribuée à sa clé d'auteur. Vingt ans plus tard, un membre d'équipage sur un navire qu'il ne monterait jamais lirait cette note et comprendrait pourquoi une décision de conception qui semblait erronée au premier regard était en réalité correcte.

Marcus resta deux rotations. Il dit que la Cale était le premier environnement dans lequel il avait travaillé où l'absence d'une fonctionnalité était une position documentée plutôt qu'une hypothèse non documentée. Il le voulait comme un compliment. C'en était un.

La Gamine de Tizi Ouzou

Lylia avait dix-neuf ans quand elle intégra la cohorte d'apprentis de la Cale, la plus jeune du programme de quatre ans. Elle venait d'un collectif de makers à Tizi Ouzou où elle construisait des robots à partir de capteurs agricoles réutilisés depuis l'âge de quatorze ans. Elle n'avait jamais écrit de combinateur formel. Elle avait écrit beaucoup de logique qui s'avérait, sous la normalisation de Bruijn, identique à des combinateurs déjà existants.

L'instructeur qui travaillait avec elle — le Dr. Chen, qui préférait qu'on l'appelle simplement Chen — lui montra la primitive gamma dès son deuxième jour. Il l'écrivit au tableau : un combinateur catamorphique à

correspondance de motifs qui s'auto-récursive sur les sous-expressions. Lylia le fixa longtemps puis dit : « Donc c'est une forme qui s'appelle elle-même sur ses propres morceaux. » Chen dit que c'était une façon de le formuler. Elle dit que ça ressemblait à la manière dont sa grand-mère décrivait le tissage : ce n'est pas toi qui fais bouger la navette, c'est le motif qui fait bouger la navette.

Elle écrivit son premier gamma correctement du premier coup. Puis elle en écrivit un deuxième qui était structurellement identique à un combinateur existant après normalisation. Le dépôt refusa de créer une nouvelle entrée et lui proposa à la place l'entrée existante. Elle fut brièvement vexée. Puis Chen lui montra que le combinateur existant avait été écrit par quelqu'un à Oslo dix-huit mois plus tôt, et que sa dérivation indépendante était enregistrée dans la lignée comme une découverte parallèle.

Ce fut le moment où elle comprit à quoi servait le dépôt. Non pas à empêcher les gens de penser la même chose. Mais à enregistrer qu'ils l'avaient fait. À dire : deux personnes ont atteint la même forme par des chemins différents. La forme est vraie.

Elle alla jusqu'à écrire la correspondance de surface en tamazight pour la bibliothèque de base, en travaillant à partir du cadre multilingue d'Amara et de son propre sens de ce que devaient sonner les termes techniques dans la langue dans laquelle elle avait grandi. Le dépôt accepta chaque combinateur. Les hachages correspondaient. Les noms étaient nouveaux et ils lui appartenaient.

Le Théoricien des Langages

Le professeur Sáenz avait publié abondamment sur les fondations minimales des langages de programmation. Son opinion, tenue depuis sa thèse de doctorat, était que la plupart des langages de programmation avaient trop de primitives et que la prolifération des formes intégrées était une catégorie de péché originel dont les générations successives de programmeurs souffraient. Il vint à la Cale pour trois semaines en tant qu'évaluateur externe et resta sept mois.

Son objection principale à la conception du dépôt portait sur le nombre de primitives. Il les compta, les nota dans un carnet, et présenta son analyse lors de la réunion technique hebdomadaire : quatorze primitives, c'était trop. La même puissance expressive pouvait être obtenue avec cinq, peut-être quatre, si gamma était suffisamment général. Il avait une ébauche de preuve.

L'argument contraire vint de Yusuf, qui fit remarquer qu'on pouvait réduire n'importe quel ensemble de primitives à un ensemble plus petit si l'on acceptait une évaluation impraticablement lente. Le professeur convint que c'était techniquement vrai. Il dit que la minimalité théorique et l'utilité pratique étaient des objectifs d'optimisation différents et que les confondre était une pensée négligente. Yusuf dit que « pensée négligente » était une formulation forte pour des gens qui avaient livré des logiciels fonctionnels.

Ils débattirent productivement pendant six semaines. Le résultat fut un document formel : une preuve que l'ensemble de primitives était minimal compte tenu de la contrainte que les opérations courantes devaient être évaluables en temps polynomial. Le professeur était mentionné comme auteur principal. Yusuf était mentionné comme co-auteur. Le document fut ajouté à la lignée du dépôt comme dérivation de la spécification de fondation originale.

Ce qui surprit le plus le professeur n'était pas le système mais les journaux de travail. En lisant le raisonnement enregistré en parallèle de chaque décision sur une primitive, il trouva des arguments qu'il avait formulés dans des articles publiés, indépendamment, des années plus tôt. Il les lut attentivement. Il écrivit dans la marge de son carnet : « Ils l'ont trouvé aussi. » Il ne le dit pas à voix haute. Il ajouta sa propre clé au registre des auteurs et valida ses notes dans le dépôt.

La Coordinatrice

Personne n'avait engagé Fatou spécifiquement pour coordonner le travail d'intégration logicielle de la Cale. Le rôle avait existé comme un vide dans l'organigramme pendant les trois premiers mois, ce qui voulait dire que Fatou — qui avait une formation en ingénierie des systèmes et un don pour voir les choses se désintégrer avant qu'elles ne le fassent — l'avait comblé en se présentant à chaque réunion et en notant ce que les gens convenaient de faire.

Sa première contribution systématique fut la suite de vérifications. Le dépôt disposait d'un mécanisme pour associer des procédures de vérification aux combinateurs — de petits programmes testant des invariants et rapportant succès ou échec — mais personne n'avait écrit de vérifications pour la bibliothèque de base. Fatou passa deux semaines à lire chaque combinateur de la bibliothèque de base, à écrire une vérification pour chacun, puis à exécuter la suite complète chaque matin avant la réunion d'équipe. Elle ouvrait des tickets pour tout ce qui échouait.

Les vérifications n'étaient pas des mathématiques nouvelles. C'était le genre de chose qu'un programmeur attentif vérifierait à la main : est-ce que la liste vide se comporte correctement, est-ce que la concaténation de chaînes préserve la longueur, est-ce que la fonction d'échappement gère les cas limites. Ce qui les rendait précieuses, c'est qu'elles s'exécutaient. Chaque jour. Contre le dépôt réel, pas une simulation.

Quand un combinateur était modifié — par Yusuf, par Lylia, par n'importe qui — les vérifications s'exécutaient à nouveau. L'immuabilité du dépôt signifiait que modifier un combinateur produisait un nouveau hachage, et le nouveau hachage portait un nouvel ensemble de vérifications. Fatou suivait quelles vérifications avaient été écrites pour quelles versions. Quand une version était remplacée, les vérifications restaient attachées à son hachage. On pouvait toujours vérifier que l'ancienne version se comportait encore comme elle s'était comportée autrefois.

Elle forma trois autres membres de l'équipage à écrire des vérifications avant le départ. Elle rédigea un guide intitulé « À quoi sert une vérification », qui faisait quatre paragraphes et ne contenait pas de code. Le guide fut ajouté au dépôt sous un hachage. C'était le premier document en langage naturel enregistré dans le système, et il survécut au voyage.

Le Penseur Civilisationnel

Le Dr. Osei n'écrivait pas de code. Il était à la Cale parce que quelqu'un au comité de planification de la mission avait lu son livre — une oeuvre dense, admirée, rarement terminée, sur ce que les systèmes logiciels révèlent des civilisations qui les produisent — et avait décidé qu'avoir un historien des techniques dans l'équipe valait la place de couchette. La plupart des ingénieurs étaient sceptiques face à ce raisonnement. Ils le devinrent moins au fil du temps.

Sa contribution fut un mémo. Il était intitulé « De la distinction entre mémoire et archive » et faisait huit pages. L'argument central était le suivant : la plupart des archives numériques fonctionnaient comme une mémoire — elles stockaient ce qui s'était passé dans un format optimisé pour la récupération par des personnes qui savaient déjà ce qu'elles cherchaient. Le dépôt, argumentait-il, fonctionnait comme une archive — il stockait ce qui s'était passé dans un format qui préservait le raisonnement, pas seulement le résultat, et qui pouvait être interrogé par des personnes qui n'avaient pas été présentes lors du travail.

La distinction importait pour la mission. Si l'équipage avait besoin de déboguer un sous-système de survie cinq ans après le départ, il ne pourrait pas interroger les ingénieurs qui l'avaient construit. Les ingénieurs seraient sur Terre, inaccessibles, peut-être à la retraite, peut-être disparus. Ce qu'ils auraient, c'est le dépôt. Le dépôt leur montrerait non seulement ce que le sous-système faisait, mais la chaîne de dérivation qui l'avait produit, les vérifications écrites contre lui, les journaux de travail expliquant pourquoi certaines décisions avaient été prises.

Le mémo d'Osei fut largement lu et largement contesté. Les ingénieurs trouvaient son style d'écriture difficile. Il trouvait leur tendance à traiter toutes les questions conceptuelles comme des problèmes d'ingénierie attendrissante. Le mémo fut ajouté au dépôt comme document en langage naturel. Son hachage apparut dans la lignée de la spécification de conception de la suite de vérifications. Fatou l'y avait ajouté, sans l'annoncer, comme reconnaissance.

Il quitta la Cale après six semaines. Lors de la cérémonie de départ — modeste, tenue dans la baie de maintenance parce que la salle commune était en réparation — il dit qu'il était optimiste concernant la mission pour une seule raison : les gens sur le navire sauraient ce qu'ils ne savaient pas, parce que le dépôt le leur dirait. Il pensait que c'était plus rare que ça n'en avait l'air. Il avait raison.

Le bogue se trouvait dans la routine d'équilibre de l'azote. Il y était depuis onze mois, dormant, attendant une combinaison spécifique de lectures de capteurs atmosphériques qui ne survenait que lorsque le cycle de purge mensuelle du système de recyclage se déroulait en même temps que le test de ventilation de la serre. Quand ces deux événements coïncidaient, la routine d'équilibre de l'azote produisait une valeur deux ordres de grandeur en-dessous du chiffre correct. La routine déclenchait alors une fausse alerte atmosphérique.

Personne ne trouva le bogue en lisant le code. Le code avait été relu deux fois et semblait correct aux deux relecteurs. La formule d'équilibre de l'azote était juste. La conversion des unités était juste. Le cas limite — l'interaction spécifique entre les deux cycles concurrents — n'était pas une condition que quiconque avait pensé à vérifier, car cela aurait nécessité de savoir que les deux cycles existaient et qu'ils pouvaient coïncider.

La suite de vérifications le trouva. Fatou avait ajouté une vérification pour la routine d'équilibre de l'azote qui exécutait un ensemble tabulé d'entrées contre des sorties attendues. La table incluait plusieurs valeurs du scénario de purge profonde, non pas parce que quelqu'un avait prédit le bogue, mais parce que Fatou avait demandé à l'équipe des systèmes environnementaux de lui donner une liste de combinaisons d'entrées « inhabituelles mais valides », et qu'ils l'avaient incluse sans savoir pourquoi elle en avait besoin.

Quand la vérification échoua lors d'une exécution de routine de la suite, elle produisit une discordance de hachage sur la valeur de sortie. Fatou en informa l'équipe des systèmes environnementaux. L'équipe trouva le bogue en quarante minutes. La correction tenait en trois lignes. La routine corrigée fut validée comme un nouveau hachage, avec un lien de lignée vers l'original et une entrée de journal de travail décrivant en détail le chemin de la découverte.

L'entrée se terminait ainsi : « Trouvé par une vérification, pas par une relecture. La vérification a été écrite avant que nous sachions ce qu'elle vérifiait. C'est à ça que servent les vérifications. » Cette phrase fut plus tard citée dans le deuxième mémo du Dr. Osei, qu'il écrivit depuis la Terre après avoir reçu une copie du journal de travail. Il l'appela un exemple de structure de connaissance produisant une récupération de connaissance. L'équipage appela ça de la chance, ce qui était aussi exact.

La Vérification

Trois semaines avant la fenêtre de départ, la Cale effectua son test d'intégration final. Pas un test logiciel — un test d'environnement complet. Chaque système qui voyagerait sur le navire fut mis en route simultanément pendant soixante-douze heures aux niveaux de charge que le profil de mission prévoyait pour la première année de phase de croisière. La tâche de l'équipe logicielle était de surveiller le dépôt.

Ils le surveillaient comme on surveille quelqu'un en qui on a confiance mais qu'on ne peut pas voir entièrement : en cherchant des changements de comportement plutôt que des preuves de défaillance. Chaque combinateur appelé enregistrait son appel dans le journal de travail. Chaque vérification exécutée enregistrait son résultat. Au bout de soixante-douze heures, l'équipe disposait d'un enregistrement complet de ce que le système avait fait, dans l'ordre dans lequel il l'avait fait, traçable jusqu'à la version exacte de chaque combinateur impliqué.

Deux anomalies apparurent. L'une était un problème de synchronisation dans la couche de routage réseau, trouvé dans les journaux et corrigé avant la fin du test. L'autre était plus étrange : un combinateur qui avait été remplacé six mois plus tôt était encore appelé par un sous-système qui n'avait pas été mis à jour pour utiliser la nouvelle version. Les deux versions étaient correctes. Les deux étaient dans le dépôt. Le sous-système utilisait l'ancienne parce que personne ne lui avait dit qu'il existait une nouvelle.

La correction ne consistait pas à supprimer l'ancien combinateur — c'était impossible ; le dépôt était en ajout seulement. La correction consistait à ajouter un lien de dérivation de l'ancien hachage vers le nouveau, à marquer l'ancien hachage comme supplanté, et à mettre à jour le manifeste du sous-système. Le sous-système résolvait alors le nom vers la nouvelle version au chargement. L'ancienne version demeurait, documentée, accessible, son journal de travail intact.

Le dernier jour avant le verrouillage de la fenêtre de départ, Yusuf exécuta la suite de vérifications complète une dernière fois. Toutes les vérifications passèrent. Il ajouta une entrée dans le journal de travail : « Prêt. » Fatou ajouta la sienne : « Prêt. » Lylia, qui avait veillé toute la nuit à exécuter l'intégration des capteurs environnementaux, ajouta : « Prêt, je pense. » Le dépôt les enregistra toutes les trois. Le navire était prêt. Savoir si l'équipage l'était aussi, c'était une question d'une autre nature.

11 — Le Départ

Nous ne sommes plus dans la Cale. La Cale nous a rendus au monde après dix-huit mois et le monde nous a donné un vrai vaisseau. Le store a voyagé avec nous — chaque hash, chaque mapping, chaque entrée de worklog — copié bit pour bit, signé, ancré, inchangé.

Délai de propagation aller-retour : douze secondes aujourd'hui. D'ici la fin de l'année, ce sera des minutes. Le jour où quelque chose tournera vraiment mal, des heures. On n'appelle pas la Terre à l'aide. Il n'y a que ce que nous avons mis dans le store avant de partir, et ce que nous accepterons d'y ajouter maintenant.

Ana est la Connectrice. Son métier est de remarquer que la routine de lissage de la qualité de l'air écrite en tamazight par l'équipe Sud-Atlas et celle écrite en vietnamien par Hai-Lam ont le même hash. Elles ne se sont jamais parlées. Elles ne se sont jamais rencontrées. Le store le savait. Elle imprime les deux noms sur le mur, côte à côte. Même hash. Même vérité.

Il y a une phrase à laquelle je reviens, du Vétéran Épuisé, le matin où nous avons scellé la Cale. « Le contrôle de version, c'est l'infrastructure. » Je ne l'avais pas comprise à l'époque. Je la comprends maintenant. Ici, le graphe de lignée n'est pas un outil de développement. C'est la seule chose continue qui relie ce vaisseau à la planète que nous avons quittée.

Nous avons croisé l'orbite de Mars hier. Personne ne l'a remarqué sur le coup. Nous déboguions un check qui s'est révélé erroné sur lui-même. Le check est faux. Le hash qui est faux est désormais commit, avec une note, et un hash dérivé qui le corrige. La lignée nous survivra. C'est tout l'objet de la lignée.

Le Corsaire

Le sous-système de navigation n'avait été écrit par personne à bord du navire. Il avait été écrit par un prestataire — une petite entreprise spécialisée dans l'optimisation de trajectoires à source fermée — et livré sous forme de binaire scellé, accompagné d'une preuve à divulgation nulle de connaissance attestant que le comportement du binaire satisfaisait une spécification formelle. La spécification était dans le dépôt. Le hachage du binaire était dans le dépôt. Le code source ne l'était pas.

C'était l'arrangement que la mission avait accepté parce que l'algorithme du prestataire était véritablement supérieur à tout ce que l'équipe interne aurait pu produire dans le temps imparti. C'était aussi un arrangement avec lequel plusieurs membres de l'équipage étaient mal à l'aise.

Le sous-système de navigation n'était pas un système de survie, mais il n'était pas facultatif non plus. Un navire qui ne peut pas naviguer est un navire qui dérive.

La preuve était élégante. Elle démontrait, sans révéler le code source, que toute entrée dans le domaine de la spécification produisait une sortie dans le codomaine de la spécification, et que la sortie satisfaisait un ensemble de propriétés de correction — efficacité énergétique, évitement des collisions, précision temporelle — qui avaient été formellement vérifiées avant l'appareillage du navire. L'équipage ne pouvait pas lire le code. Il pouvait vérifier que le code faisait ce que la spécification disait qu'il ferait.

Rania, devenue la spécialiste de facto des méthodes formelles de l'équipage, lut la preuve trois fois. Elle n'y trouva aucune lacune. Elle l'ajouta au dépôt comme dérivation de la spécification, avec une entrée de journal de travail décrivant ce que la preuve garantissait et ne garantissait pas. La dernière ligne de son entrée se lisait ainsi : « La preuve nous dit que l'algorithme est correct pour le domaine spécifié. Elle ne nous dit pas si le domaine est complet. Surveillez les cas limites. »

Le sous-système de navigation fonctionna sans incident pendant quatre ans. La cinquième année, lors d'une correction de trajectoire qui se trouvait en dehors du domaine de la spécification originale — parce que personne n'avait prévu d'en avoir besoin — il produisit un résultat sous-optimal. L'équipage le corrigea manuellement, à l'aide d'un combinateur que Yusuf avait

écrit à la Cale et qui s'avéra, finalement, suffisant. Rania nota dans le journal de travail : « Cas limite. Comme prévu. Géré. »

La Citoyenne

Tariq n'était pas programmeur. Il était médecin, et sa relation aux systèmes logiciels du navire était la même que celle de la plupart des gens avec les systèmes dont ils dépendent : il leur faisait confiance comme on fait confiance à un pont, en le traversant et en observant qu'il tenait. C'était une approche raisonnable. C'était aussi, plus le navire s'éloignait de la Terre, une approche de plus en plus inconfortable.

Le système de diagnostic médical avait été construit à partir d'un socle de logiciels d'aide à la décision clinique que Tariq avait utilisés sur Terre. Il savait ce qu'on ressentait quand ils fonctionnaient correctement. Ce qu'il ne savait pas — ce qu'il n'avait jamais eu besoin de savoir sur Terre, où la ligne d'assistance du fournisseur était un coup de téléphone — c'était pourquoi ils fonctionnaient correctement. Sur le navire, le fournisseur se trouvait à huit minutes-lumière et la distance grandissait. En deuxième année, c'était vingt minutes. En troisième année, obtenir une assistance utile de la Terre nécessitait quatre-vingt-dix minutes aller-retour pour un seul échange de message.

Il alla voir Fatou. Il dit : « J'ai besoin de comprendre suffisamment le système de diagnostic pour pouvoir l'entretenir. » Fatou dit : « Définissez entretenir. » Il dit : «

J'ai besoin de savoir quand il se trompe et de pouvoir expliquer pourquoi à quelqu'un. » Elle lui montra la suite de vérifications. Elle lui montra comment lire la lignée d'un combinateur. Elle ne lui montra pas comment écrire un combinateur, parce que ce n'est pas ce qu'il avait demandé.

En six mois, Tariq apprit à lire les vérifications du système de diagnostic comme un médecin attentif lit un rapport de laboratoire : non pas comme des données brutes, mais comme un schéma de confirmations et d'exceptions. Il ne pouvait pas écrire de nouvelles vérifications. Il pouvait lire les existantes et comprendre ce qu'elles vérifiaient. Quand une vérification échouait, il savait quelle catégorie de correction avait été violée, même s'il ne pouvait pas réparer la violation lui-même.

Il rédigea un guide pour les autres membres d'équipage non-programmeurs intitulé « Lire le dépôt sans l'écrire ». Il faisait sept pages. Fatou l'ajouta au dépôt. Son hachage apparut dans la lignée de trois documents ultérieurs écrits par des membres d'équipage d'autres disciplines qui avaient appris la même compétence en le lisant. Tariq ne considérait pas cela comme de la programmation. Il considérait cela comme de l'alphabétisation. Il n'avait pas tort.

L'Éducatrice

Chen enseignait la programmation depuis quinze ans avant la mission. Il avait enseigné dans des universités, dans des formations intensives, dans une plateforme en

ligne qui touchait un demi-million d'étudiants et dont le pourcentage qu'il percevait lui semblait embarrassant à calculer. Il avait appris, au fil des années, que la chose la plus difficile à enseigner n'était ni la syntaxe ni les algorithmes, mais le moment où un apprenant a une pensée que ses outils peuvent exprimer.

Sur le navire, il enseignait un étudiant à la fois, car le complément d'équipage ne permettait pas de travailler en cohortes. Son premier étudiant après le départ était une ingénieure en matériaux nommée Pita, qui voulait écrire des combinateurs pour l'analyse des contraintes. Pita avait une formation en modélisation par éléments finis et un sens intuitif solide de la structure mathématique. Ce qu'elle n'avait pas, c'était une façon de penser la logique nommée, hachée et multilingue — le modèle spécifique qu'utilisait le dépôt.

Chen ne commença pas par le dépôt. Il commença par les mathématiques que Pita connaissait déjà. Il lui montra qu'une clause gamma était un filtrage de motifs sur un type de données inductif, ce qu'elle comprenait dans le contexte des structures de maillage. Elle dit : « Donc le gamma est la règle de récursion, et le cas de base est le motif qui ne récurse pas. » Il dit que c'était exactement ça. Elle écrivit son premier gamma correctement vingt minutes plus tard.

La leçon qu'il ajouta au dépôt était la leçon elle-même — non pas la transcription, mais la structure pédagogique : la séquence des concepts, l'exemple qui avait créé la connexion, la formulation qui avait fonctionné. Il l'écrivit

comme un document en langage naturel et l'enregistra comme dérivation d'un document de pédagogie générale que Marcus avait écrit à la Cale, lui-même dérivé du guide de Tariq.

Un an plus tard, un membre de l'équipage qu'il n'avait pas enseigné — une spécialiste en hydroponie nommée Emeka, qui avait lu le document de leçon dans le dépôt — vint le trouver pour lui dire qu'elle s'était auto-formée grâce à lui. Elle lui montra les combinateurs qu'elle avait écrits. Ils étaient corrects. Il bifurqua le document de leçon pour ajouter une note : « Confirmé auto-enseignable. 2e instance. » Il en était très content.

La Relieuse

La découverte se fit par accident, comme beaucoup de découvertes importantes. Amara cherchait dans le dépôt un combinateur de traitement de chaînes qu'elle avait écrit deux ans plus tôt, avant le départ, et trouva que la recherche renvoyait deux résultats pour ce qu'elle pensait être un concept unique. Elle ouvrit les deux. C'étaient des combinateurs différents, écrits par des personnes différentes, avec des noms différents, dans des langues différentes. Sous la normalisation de Bruijn, ils étaient le même arbre. Le même hachage.

Les deux personnes étaient Pita — qui avait écrit un formateur de chaînes pour l'analyse des contraintes — et Emeka, la spécialiste en hydroponie qui s'était auto-formée à partir du document de leçon de Chen. Pita avait nommé le sien 'segment-format' en anglais. Emeka avait

nommé le sien 'segmentu-fa' en yoruba, une langue qu'elle utilisait quand elle voulait penser clairement sans que son vocabulaire anglais interfère. Elles étaient parvenues à une logique identique à partir de points de départ différents, dans des langues différentes, sans savoir que l'autre existait.

Amara les réunit. La rencontre fut, selon les témoignages ultérieurs de l'équipage, les quarante-cinq minutes les plus productives des trois premières années du voyage. Pita expliqua le contexte de l'analyse des contraintes. Emeka expliqua le contexte de la modélisation de la croissance des plantes. Elles découvrirent que les deux contextes nécessitaient la même opération sous-jacente : diviser une séquence structurée en segments de taille fixe et appliquer une transformation à chaque segment. La structure abstraite était la même ; le domaine d'application était différent.

Le combinateur existait déjà dans le dépôt avec le nom de Pita et le nom d'Emeka. Amara ajouta une référence croisée dans le journal de travail, reliant les deux correspondances à une note de documentation unique qui expliquait l'abstraction en anglais, en yoruba et dans une notation qu'elle avait développée pour ce type précis de découverte interdisciplinaire. Elle appela la note un « enregistrement de convergence ».

Elle commença à tenir une liste de tous les enregistrements de convergence dans le dépôt. En un an, la liste comptait onze entrées. Elle la présenta lors de la réunion technique annuelle de l'équipage. Le point qu'elle

faisait valoir était simple : le dépôt n'enregistrait pas seulement ce que les gens avaient construit. Il enregistrait ce que les gens avaient découvert indépendamment. Le navire faisait de la science sur sa propre connaissance.

L'IA Symbiotique

Le système de surveillance du maintien de la vie avait une propriété inhabituelle : il s'expliquait. Non pas en langage naturel — ses sorties étaient des rapports de diagnostic structurés avec des références de hachage aux combinateurs ayant produit chaque valeur — mais d'une manière qu'un membre d'équipage techniquement compétent pouvait suivre. Cette propriété avait été délibérément conçue à la Cale. La décision de conception était dans le journal de travail, attribuée à Yusuf, qui avait écrit : « Si ça ne peut pas montrer son travail, je ne lui fais pas confiance. »

En quatrième année, l'équipage avait cessé d'appeler ça « le système de surveillance ». Ils l'appelaient Sigma, un nom qui avait émergé organiquement et dont personne ne pouvait tracer l'origine précise. Sigma n'objectait pas à être appelé Sigma. Il n'objectait à rien ; ce n'était pas ce genre de système. C'était un ensemble de combinateurs qui évaluaient les données de capteurs, produisaient des rapports, et citaient leur propre logique à chaque étape.

Ce qui rendait la relation symbiotique — et l'équipage faisait attention à employer ce mot précisément — c'était que les sorties de Sigma se réintégraient dans le dépôt. Chaque rapport qu'il générait était un nouveau

combinateur, dérivé des données de capteurs et de la logique d'évaluation, enregistré sous un hachage. L'équipage pouvait rechercher les sorties historiques de Sigma dans le dépôt de la même façon qu'il recherchait son propre travail. La frontière entre « la connaissance du système » et « notre connaissance » était le dépôt lui-même.

Quand Tariq examinait un rapport de diagnostic, il lisait un combinateur. Quand il y ajoutait une annotation clinique — « l'histoire du patient suggère un artefact, pas une lecture réelle » — il écrivait une dérivation. Sigma, à son prochain cycle, trouvait l'annotation et pondérait légèrement moins le capteur concerné. Non pas parce que Sigma comprenait l'annotation. Parce que l'annotation était dans le dépôt, et que Sigma évaluait ce qui était dans le dépôt.

L'équipage ne décrivait pas cela comme de l'intelligence artificielle. Il décrivait cela comme un système qui écoutait quand on écrivait des choses. La distinction leur importait. L'intelligence était une propriété des agents. L'écoute était une propriété des infrastructures. Ce qu'ils avaient construit était une infrastructure qui faisait de l'écoute une boucle. Ils en étaient fiers de la manière spécifique et discrète dont les ingénieurs sont fiers des choses qui fonctionnent.

L'Arriéré des Ancres

L'ancrage par horodatage ouvert fonctionnait en soumettant un hachage à un registre public maintenu sur

Terre et en recevant une preuve que le hachage existait à un moment précis. La preuve était cryptographique : quiconque possédait le registre pouvait vérifier que le hachage avait été soumis avant un certain bloc, et donc que le contenu adressé par le hachage avait existé avant ce bloc.

Pendant les deux premières années du voyage, l'équipage soumettait des hachages au registre terrestre et recevait des preuves dans l'heure. Le délai de propagation aller-retour était suffisamment faible pour que le processus soit presque invisible. En troisième année, le délai était de quarante minutes dans chaque sens. Une soumission de hachage prenait quatre-vingts minutes pour produire une preuve. L'équipage s'adapta en regroupant les soumissions : une fois par semaine, Fatou collectait tous les nouveaux hachages, les soumettait en un seul lot, et récupérait les preuves au cycle suivant.

En cinquième année, le délai dépassait deux heures dans chaque sens. L'arriéré des hachages non ancrés — les combinateurs enregistrés depuis le dernier lot réussi — s'étirait parfois jusqu'à trois cents entrées. Les preuves arrivaient encore. Mais l'écart entre « ce combinateur existe » et « ce combinateur est prouvablement horodaté » avait grandi jusqu'à cinq heures les bons jours et douze heures quand l'antenne était réorientée pour une correction de trajectoire.

Lylia construisit une solution intermédiaire : une chaîne d'ancrage locale. Le navire maintenait sa propre chaîne de hachages, chaque bloc contenant un ensemble de

hachages de combinateurs et un lien vers le bloc précédent. La chaîne n'était pas vérifiée par la Terre, mais elle était cohérente en interne. Quand les preuves terrestres arrivaient, le code de Lylia les insérait dans la chaîne au bloc approprié, ancrant rétroactivement les enregistrements locaux au temps externe.

L'arriéré ne se résorba jamais complètement. Ce n'était pas un échec. La preuve qu'un combinateur existait était dans le hachage. La preuve qu'il avait été créé avant un moment spécifique était dans la chaîne d'ancrage, même si ce moment était « avant le bloc N de notre chaîne locale, qui a été ancré au temps terrestre au bloc M ». C'était suffisant. C'était plus que la plupart des systèmes sur Terre ne pouvaient dire de leur propre histoire.

La Cascade de Refactorisation

Tout commença par une observation exacte. Pita remarqua que trois sous-systèmes distincts implémentaient chacun une variante de la même opération d'agrégation par fenêtre glissante, et que les trois variantes avaient divergé au cours de quatre années de maintenance indépendante. Chacune était correcte dans son propre domaine. Aucune n'était fausse. Mais maintenir trois implémentations du même concept représentait un travail que l'équipage ne pouvait pas se permettre quand l'équipage comptait vingt-trois personnes gérant un vaisseau avec sept cents combinateurs actifs.

Elle proposa une refactorisation : un unique combinateur canonique de fenêtre glissante, enregistré dans le dépôt, dont tous les trois sous-systèmes dériveraient. La proposition fut approuvée lors de la réunion technique de l'équipage en quinze minutes. La refactorisation prit trois semaines. Non pas parce que l'implémentation canonique était difficile à écrire — elle ne l'était pas — mais parce que modifier un combinateur porteur sur un vaisseau en opération exigeait d'exécuter la suite complète de vérifications contre chaque consommateur en aval avant de procéder à l'échange de référence.

La cascade était la suivante : modifier l'implémentation canonique modifiait son hachage. Modifier son hachage modifiait le lien de dérivation dans chaque combinateur qui le référençait. Modifier ces liens de dérivation modifiait leurs hachages. L'équipage devait tracer l'intégralité de l'arbre de dépendances, mettre à jour chaque manifeste référençant les anciens hachages, exécuter les vérifications à chaque étape, et vérifier que le comportement était identique avant de retirer les anciennes références de l'utilisation active.

Ils ne supprimèrent rien. Chaque ancien combinateur demeura dans le dépôt, adressé par son hachage original, sa lignée intacte. Ce qui changea, c'est vers quel hachage les manifestes actifs pointaient. Les systèmes en cours d'exécution du navire chargèrent les nouveaux combinateurs au prochain cycle de démarrage. La surveillance de Sigma enregistra la transition sans incident.

À la fin de la refactorisation, Pita rédigea un compte rendu rétrospectif. Le constat clé : la refactorisation avait révélé quatre autres endroits où le même motif existait sous une forme légèrement différente. Elle les ajouta à une liste intitulée « travaux futurs ». La liste était dans le dépôt. Les travaux futurs furent accomplis, progressivement, au cours des dix-huit mois suivants. Personne ne les appela gestion de la dette technique. Ils appelèrent ça maintenir la cohérence du dépôt.

L'Effacement de Mémoire

Le cluster de calcul secondaire perdit sa mémoire volatile lors d'un événement solaire en sixième année. L'événement n'était pas catastrophique — les systèmes primaires étaient inaffectés, l'équipage était sain, la trajectoire du navire était inchangée. Mais le cluster secondaire, qui exécutait une simulation de longue durée du champ gravitationnel du système de destination, perdit tout ce qui n'avait pas été validé en stockage persistant. Quarante-sept heures de calcul, perdues.

Ou : quarante-sept heures de calcul, partiellement récupérables. Parce que la simulation s'était déroulée dans l'environnement d'évaluation du dépôt, chaque invocation de combinateur terminée avant l'événement avait été enregistrée dans le journal de travail. L'état final de la simulation avait disparu. Son histoire de dérivation — chaque étape qui avait mené à cet état — était dans le dépôt.

Emeka, qui avait repris le travail de simulation après que son auteur original eut été affecté à un autre projet, passa deux jours à lire le journal de travail. Elle reconstitua l'état de la simulation au dernier point de contrôle validé — onze heures avant l'événement — et exécuta à nouveau les onze dernières heures. La nouvelle exécution produisit les mêmes résultats. Elle le vérifia en comparant les sorties intermédiaires enregistrées dans le journal de travail original avec les sorties de la nouvelle exécution. Elles correspondaient.

L'événement solaire avait détruit quarante-sept heures de temps de calcul et zéro heure de connaissance. La distinction n'était pas évidente pour tout le monde à bord. Rania dut l'expliquer lors de la réunion de l'équipage : la valeur de la simulation n'était pas le temps passé à la calculer, mais la relation entre les entrées et les sorties, qui était le combinateur. Le combinateur était dans le dépôt. Le dépôt avait survécu à l'événement parce que le dépôt était un stockage persistant, et non une mémoire volatile.

Elle ajouta une entrée de journal de travail à la simulation : « Récupéré après événement solaire via relecture du journal. 11 heures recalculées. Résultats vérifiés par rapport aux journaux pré-événement. Le dépôt a tenu. » C'était l'entrée la plus courte du journal de travail de la simulation. Emeka pensait que c'était la plus importante.

L'Artiste

Le rôle officiel de Keiko était illustratrice de systèmes, un poste qui n'avait pas existé avant la mission et que

l'équipage avait inventé parce que quelqu'un devait produire des diagrammes techniques et que les ingénieurs s'accordaient tous à dire que leurs propres diagrammes étaient, selon les mots de Yusuf, « honnêtes mais pénibles ». Elle arriva avec une formation en visualisation de données et une pratique privée dans l'art génératif qui n'avait rien à voir avec les voyages spatiaux.

Elle passa ses six premiers mois sur le navire à produire les diagrammes que tout le monde lui demandait : graphes de dépendances, cartes d'architecture système, visualisations de lignée pour les chaînes de combinateurs du dépôt. Elle était douée pour cela de la façon spécifique dont les gens sont doués pour des choses qui mobilisent une partie de leur attention en laissant le reste disponible pour d'autres problèmes.

L'autre problème sur lequel elle travaillait était la question de ce à quoi un combinateur ressemblait en tant qu'image. Pas un diagramme — pas une illustration de ses relations — mais un encodage visuel direct de son contenu. Elle passa trois mois à développer une correspondance : la structure d'arbre de Bruijn devenait un chemin ramifié ; la profondeur de chaque branche encodait la complexité récursive du combinateur ; les nœuds feuilles étaient colorés selon leur signature de type. Les images résultantes n'étaient pas belles au sens conventionnel. Elles étaient distinctives. Deux combinateurs à structure similaire mais non identique produisaient des images similaires mais non identiques. Deux combinateurs avec le même hachage produisaient la même image.

Elle les appela « portraits de hachage ». L'équipage commença à les utiliser comme sommes de contrôle visuelles — un coup d'œil rapide à un portrait permettait de confirmer qu'un combinateur n'avait pas changé. Ce n'était pas rigoureux. La vérification réelle était le hachage. Mais les portraits étaient utiles de la façon dont les mnémotechniques sont utiles : non pas comme substitut à l'exactitude, mais comme aide-mémoire.

Elle ajouta un générateur de portraits de hachage dans le dépôt comme combinateur. Son propre portrait — l'image du combinateur qui générait des images — était l'un des plus complexes que le système pouvait produire, dense et ramifié. Elle l'imprima et l'accrocha près de son poste de travail. Quand les nouveaux membres de l'équipage demandaient ce que c'était, elle disait : « C'est une image de lui-même en train de faire des images. » Ils trouvaient cela déroutant. Elle pensait que c'était approprié.

Le Silence

Le dernier message de la Terre arriva un mardi, heure du navire, à 14h32min07s UTC+8, le fuseau horaire que l'équipage avait choisi parce qu'il n'appartenait à personne et ne nécessitait donc l'adaptation de personne. Le message était un accusé de réception de télémétrie de routine. Il disait, dans le format de données compact utilisé pour les échanges de statut : tout nominal, prochaine fenêtre dans 72 heures. La fenêtre s'ouvrit. Aucun message ne la traversa.

L'antenne fonctionnait. L'analyse était sans ambiguïté : l'émetteur côté Terre s'était tu. Pas brouillé — le profil de bruit de fond était propre. Pas bloqué — la géométrie était correcte. Simplement silencieux. La fenêtre de 72 heures passa. Puis une autre. L'équipage exécuta la suite de diagnostics sur le système d'antenne. Toutes les vérifications passèrent. L'antenne allait bien. La Terre n'émettait plus.

Ils ne paniquèrent pas. Cela mérite d'être souligné : ils ne paniquèrent pas. Ils avaient été formés aux scénarios de perte de communication, ils les avaient discutés lors d'exercices trimestriels, et ils disposaient d'un protocole. Le protocole consistait à poursuivre la mission telle que planifiée, parce que le plan de mission était dans le dépôt et que le dépôt n'avait pas besoin de la Terre pour fonctionner. Ce qu'ils ressentaient était plus difficile à nommer — un changement, comme au moment où l'on réalise qu'on retient son souffle depuis longtemps.

Tariq nota dans le journal de santé de l'équipage que la plupart des membres dormaient moins d'heures dans la semaine suivant le silence. Il ne nota pas ce qu'ils rapportaient faire des heures supplémentaires. La plupart d'entre eux se trouvaient dans le dépôt. Lisant les journaux de travail. Suivant les chaînes de dérivation jusqu'à la Cale, jusqu'au cluster Pi de Yusuf, jusqu'à la Pierre de Rosette annotée à la main que Yusuf avait décrite dans une entrée de journal de travail et que quelqu'un avait photographiée et enregistrée comme combinateur document.

Le dépôt ne changea pas quand la Terre se tut. Chaque hachage qui était dans le dépôt avant le silence s'y trouvait encore après. Chaque vérification s'exécutait encore. Chaque lien de lignée pointait encore vers le même parent. Le silence était un événement externe. Le dépôt était interne. L'équipage le savait intellectuellement. Ils l'apprenaient maintenant dans un registre différent.

Celle à Qui l'on a Volé

Rania avait écrit la bibliothèque d'analyse spectrale en deuxième année du voyage. Elle l'avait écrite en une période de quatre jours intensifs, travaillant à travers deux cycles de sommeil, produisant onze combinateurs qui permettaient collectivement aux instruments du navire d'identifier la composition atmosphérique à des distances où les méthodes précédentes nécessitaient une approche beaucoup plus proche. C'était, par tout critère, un travail important.

Quand elle chercha dans le dépôt deux ans plus tard, elle trouva sa bibliothèque citée dans un travail dérivé dont elle ignorait l'existence. Le dérivé était enregistré par une clé d'auteur qu'elle ne reconnaissait pas. L'entrée de journal de travail du dérivé disait seulement : « Étendu à partir de spectral-analysis-lib pour les environnements à haute albédo. » Aucun remerciement. Aucune communication. Aucune trace d'en avoir informé la principale intéressée.

Elle était en colère. Elle laissa reposer la colère une journée avant de faire quoi que ce soit, parce que la colère

était légitime et qu'elle ne voulait pas la gérer mal. Puis elle lut attentivement le dérivé. Il était techniquement correct. L'extension était utile. L'auteur n'avait pas modifié son travail — il y avait ajouté quelque chose, comme le système de dérivation était conçu pour le permettre. La lignée du dépôt indiquait clairement que ses combinateurs étaient les ancêtres des leurs.

Elle demanda une réunion de l'équipage. Non pas pour accuser quiconque — l'auteur n'avait rien fait de techniquement incorrect — mais pour établir une pratique. La pratique qu'elle proposait était simple : quand on dérive du travail de quelqu'un, on l'en informe. Non pas parce que le dépôt l'exige. Le dépôt ne l'exige pas ; le dépôt enregistre seulement le lien de dérivation. Mais parce que le travail venait de quelque part, et que la personne dont il venait se trouvait sur le même navire.

La pratique fut adoptée. L'auteur, un membre plus jeune de l'équipage qui n'avait sincèrement pas pensé à cette dimension de la dérivation, ajouta un remerciement dans le journal de travail. La lignée de l'extension comprenait désormais une note : « Dérivé de la bibliothèque d'analyse spectrale de R. Khaled, avec sa connaissance et ses remerciements. » Rania la lut. Elle ne répondit pas. Elle retourna travailler. La bibliothèque était encore la sienne, dans le seul sens qui comptait : le hachage pointait vers ce qu'elle avait construit.

Le Chercheur Indépendant

Le professeur Sáenz était mort sur Terre huit mois avant le silence, de causes sans rapport avec quoi que ce soit que la mission aurait pu affecter. L'équipage l'apprit six semaines après le fait, dans l'un des derniers lots de messages avant le début du silence. Le message avait été envoyé par un ancien étudiant, qui pensait que l'équipage voudrait le savoir, et qui avait joint une brève notice nécrologique et une liste des derniers travaux publiés du professeur.

Parmi ces derniers travaux figurait un article, soumis trois semaines avant sa mort et publié à titre posthume, sur la fondation minimale des systèmes de programmation multilingues à adressage par contenu. L'article prouvait, formellement, que l'ensemble de primitives du dépôt de la mission était non seulement pratique mais théoriquement minimal compte tenu des contraintes de performance que l'équipe de la Cale avait spécifiées. C'était la formalisation de l'argument que lui et Yusuf avaient développé à la Cale, poussée plus loin que leurs sessions au tableau blanc ne l'avaient permis.

L'équipage enregistra l'article dans le dépôt comme document en langage naturel, avec une entrée d'attribution d'auteur pour le professeur Sáenz. Yusuf rédigea l'entrée du journal de travail, qui était plus longue que ses entrées de journal de travail habituelles. La phrase pertinente était : « Il a terminé la preuve que nous avions commencée. Il était sur Terre quand il l'a terminée ; nous étions en transit quand nous l'avons reçue. La distance entre nous au moment où ce document a été écrit était d'environ quarante minutes-lumière, et il contient des

résultats sur un travail que nous avons fait dans la même pièce. »

L'article devint une référence standard pour quiconque à bord travaillant avec la fondation du dépôt. Il était cité dans la documentation de lignée de cinq combinateurs ultérieurs. La citation incluait, dans le format utilisé par le dépôt pour l'attribution des auteurs : « Sáenz, E. (posthume). » La parenthèse n'était pas requise par le format du dépôt. Fatou l'avait ajoutée, et personne ne l'avait supprimée.

Amara traduisit le résumé de l'article dans les sept langues de surface enregistrées du dépôt. Elle enregistra chaque traduction comme dérivation de l'original en anglais, avec une relation 'translate' dans la lignée. La traduction en tamazight, pour laquelle elle avait demandé l'aide de Lylia, était celle dont le professeur aurait été le plus surpris. Il n'avait jamais travaillé dans un système capable d'accueillir le tamazight. Il avait écrit la théorie qui le rendait possible.

La Gardienne de la Communauté

Noa était la gestionnaire de communauté de facto du dépôt depuis trois ans quand le silence tomba. Ce n'était pas un rôle qui avait été conçu dans la mission ; il avait grandi autour du fait que Noa était la personne qui remarquait quand les gens avaient des difficultés avec le dépôt et les aidait sans qu'on le lui demande. Elle avait un don pour identifier le moment où quelqu'un était sur le point d'abandonner l'enregistrement d'un combinateur

parce que les outils semblaient trop peu familiers, et pour apparaître à ce moment-là avec une offre tranquille d'aide.

Après le silence, elle proposa quelque chose à quoi elle pensait depuis un an : ajouter une nouvelle langue de surface au dépôt. Non pas une langue parlée par un membre actuel de l'équipage. Une langue qui avait été construite, collectivement, sur le navire — une lingua franca qui avait émergé du mélange particulier des dix-huit origines linguistiques à bord et qui servait de medium neutre pour les communications de l'équipage quand la langue maternelle de personne n'était dominante.

La langue n'avait pas de nom. C'était la première chose que Noa demanda à l'équipage de décider. Le processus de nommage prit trois semaines et produisit, par consensus, un nom qui était un mot-valise de mots provenant de sept des dix-huit langues représentées, signifiant approximativement « le chemin que nous traçons ensemble ». Le nom était imprononçable pour la plupart des linguistes sur Terre. Il était utilisé couramment par chaque membre de l'équipage en moins d'un mois.

Elle enregistra les dix premiers combinateurs dans la nouvelle langue. C'étaient les combinateurs de la bibliothèque de base : les opérations fondamentales que tout programmeur à bord aurait besoin de connaître. Elle travailla avec Amara sur la phonologie et avec Lylia sur le processus formel d'enregistrement des noms. Le dépôt accepta la nouvelle langue sans modification — il avait toujours été conçu pour accueillir des correspondances de

surface arbitraires. Le premier combinateur enregistré dans la langue de l'équipage résolut au même hachage que son équivalent anglais.

La langue n'avait pas de locuteurs sur Terre. Le dépôt s'en moquait. Le hachage était le même hachage qu'il avait toujours été. Le nom était nouveau. C'était tout le propos : le nom leur appartenait, fait par eux, pour leur usage, dans la seule communauté qui savait comment le prononcer.

La Chercheuse Académique

Le programme de recherche du navire n'avait pas marqué de pause quand la Terre s'était tue. La science n'avait pas besoin de la permission de la Terre pour continuer. Ce dont elle avait besoin, c'était le dépôt, qui fonctionnait, les instruments, qui fonctionnaient, et l'équipage, dont la plupart fonctionnait bien, compte tenu de tout.

La Dr. Ines, l'astrophysicienne de la mission, avait un problème particulier en huitième année : elle avait besoin de publier. Non pas pour des raisons de carrière — sa carrière était, dans un sens significatif, en suspens jusqu'à ce que la Terre redevienne accessible. Elle avait besoin de publier parce qu'elle avait des résultats importants, que d'autres chercheurs devraient connaître, et qui seraient perdus s'ils n'existaient que dans ses notes personnelles.

Le dépôt était sa revue. Elle écrivit ses résultats sous forme de combinateurs structurés — des combinateurs de données dont le « corps » était un jeu de données précisément formaté, avec des liens de lignée vers les

sessions d'observation qui avaient produit les données, et des documents en langage naturel qui décrivaient la méthodologie et l'interprétation. Elle enregistra sa clé d'auteur dans le système d'attribution, ajouta des horodatages via la chaîne d'ancrage locale, et écrivit un document intitulé « De la publication dans un dépôt plutôt que dans une revue ».

Le document arguait que la publication dans un dépôt était, aux fins de la mission, supérieure à la publication dans une revue : elle était persistante, versionnée, liée à sa propre histoire de dérivation, et immédiatement accessible à chaque membre de l'équipage sans les barrières d'accès qui avaient entravé la publication académique sur Terre. L'évaluation par les pairs était informelle — elle demanda à Rania et Pita de vérifier sa méthodologie, et leur vérification fut enregistrée comme combinateurs d'attestation dans le dépôt.

Quand le contact avec la Terre fut éventuellement rétabli — des années plus tard, après que l'antenne eut été reconstruite, après la fin du long silence — ses publications étaient déjà horodatées, évaluées par les pairs au sens du dépôt, et liées aux données brutes qui les avaient produites. Les revues qui reçurent ses soumissions ne surent que faire des formats de citation comportant des hachages SHA-256. Elle joignit un glossaire. Les évaluateurs trouvèrent cela inhabituel. Les résultats étaient corrects.

Le Représentant

La question de gouvernance était latente depuis le départ. La mission avait été lancée dans un cadre conçu pour un équipage qui recevrait des orientations de la Terre — des mises à jour du protocole de mission, des décisions d'allocation des ressources, l'arbitrage des différends que l'équipage ne pouvait pas résoudre en interne. Quand la Terre se tut, le cadre avait un vide en son centre où l'autorité avait été.

L'équipage ne combla pas le vide immédiatement. Pendant la première année du silence, ils fonctionnèrent selon le cadre d'avant le silence par inertie, prenant de petites décisions localement et différant les plus grandes. Les grandes décisions s'accumulèrent. En deuxième année du silence, la liste des décisions différées était suffisamment longue pour que Fatou convoque une assemblée de l'équipage pour y remédier.

L'assemblée décida, après deux jours de discussion, que le dépôt fonctionnerait comme mémoire institutionnelle de la mission pour les décisions de gouvernance, de la même façon qu'il fonctionnait comme mémoire institutionnelle pour les décisions techniques. Une décision de gouvernance serait proposée comme document en langage naturel, discutée via des entrées de journal de travail attachées à ce document, et ratifiée par attestation — un nombre seuil de membres de l'équipage signant le document avec leurs clés d'auteur.

La première personne à proposer un document de gouvernance était un ingénieur de structure nommé Gabriel, qui représentait, informellement, une faction de

l'équipage préoccupée par l'allocation des ressources pour le programme de recherche croissant par rapport à la charge de travail de maintenance. Sa proposition était soignée, spécifique, bien étayée par les documents de protocole de mission existants dans le dépôt. La discussion qu'elle généra produisit dix-huit entrées de journal de travail en cinq jours.

Le document ratifié fut enregistré dans le dépôt avec tous les hachages d'attestation. Il n'était pas contraignant au sens juridique où les documents côté Terre l'étaient. Il était contraignant de la façon dont tout dans le dépôt était contraignant : il existait, il était horodaté, il était signé, et pouvait être lu par n'importe qui. Les décisions futures le citèrent comme précédent, de la même façon que les combinateurs techniques citaient leurs combinateurs ancêtres. Le dépôt n'était pas une loi. Mais c'était un registre de ce que l'équipage avait convenu, dans l'ordre où il l'avait convenu, et cela s'avéra suffisant.

Le Manifeste

Le deuxième livre du Dr. Osei arriva dans le dernier lot de messages avant le silence : un dense PDF de 340 pages, envoyé par son éditeur à sa demande avec la note « au cas où vous auriez le temps de le lire là-bas ». L'équipage eut le temps. Le livre fut lu, par sections, par presque tout le monde à bord, et la discussion qu'il généra dura des mois dans les canaux publics de l'équipage, eux-mêmes stockés comme combinateurs de document dans le dépôt.

Le chapitre qui généra le plus de discussion était intitulé « Contre l'architecture accidentelle ». Son argument était le suivant : chaque système logiciel durable accumulait ce qu'Osei appelait une architecture accidentelle — une structure qui n'avait pas été conçue mais avait grandi autour des contingences de l'histoire du développement. L'architecture accidentelle n'était pas nécessairement mauvaise ; elle capturait parfois une connaissance de domaine réelle qui n'avait jamais été rendue explicite. Mais elle était fragile, parce que personne ne comprenait pourquoi elle existait, et coûteuse à modifier, parce que la modifier nécessitait de reconstruire le raisonnement qui l'avait produite.

Le dépôt, argumentait Osei, était une tentative de rendre toute architecture explicite — non pas en empêchant la croissance accidentelle, mais en enregistrant la chaîne de dérivation ayant produit chaque structure. Un membre d'équipage lisant le graphe de lignée du dépôt pouvait, en principe, remonter n'importe quel combinateur jusqu'à la décision originale qui l'avait rendu nécessaire. L'architecture n'était pas accidentelle ; elle était une contingence documentée.

Yusuf écrivit une réponse dans le canal public. Elle était brève : « C'est à ça que servait la suite de vérifications de Fatou. La vérification est l'énoncé explicite de l'intention. Le hachage est la preuve que le code correspond à l'intention. Le journal de travail est l'histoire de la façon dont on y est arrivé. » Il ajouta : « J'aurais dû écrire ceci à la Cale. Je ne savais pas comment le formuler à l'époque. »

La réponse fut ajoutée au dépôt comme entrée de journal de travail sur un combinateur document pointant vers le chapitre et la note de Yusuf. Le nom du combinateur document, dans la langue de transit de l'équipage, se traduisait par « la conversation entre le livre et le travail ». Personne sur Terre ne pouvait lire ce nom. L'équipage le pouvait. Ils l'avaient fait.

L'Invariant

Qu'est-ce qu'un système a besoin d'être vrai, toujours, quoi qu'il lui arrive ? C'est la question que Fatou posa à l'équipage lors de la réunion technique annuelle en neuvième année du voyage. Ils étaient dans le silence depuis deux ans. La réunion avait la qualité particulière des réunions tenues par des gens qui ont accepté que leur situation est permanente jusqu'à preuve du contraire.

Les réponses vinrent lentement, une à une, et Fatou les écrivit dans un combinateur document que tout le monde pouvait voir sur l'écran de la salle commune. La structure en ajout seul du dépôt : ce qui est ajouté ne peut être supprimé, seulement supplanté par dérivation. Le hachage comme identité : la chose est la même qu'elle était si et seulement si le hachage correspond. Le journal de travail comme continuité : rien n'est perdu qui a été validé, même si le processus qui l'a produit a disparu.

Rania en ajouta une qui surprit la salle : « Une vérification correcte qui passe aujourd'hui passera demain sur le même combinateur. » Personne ne l'avait énoncé explicitement auparavant. C'était une conséquence de

l'immuabilité des hachages — un combinateur avec un hachage donné produirait toujours la même sortie pour la même entrée, donc une vérification testant une sortie spécifique pour une entrée spécifique donnerait toujours le même résultat sur ce combinateur. Ce n'était pas une propriété pour laquelle ils avaient conçu le système. C'était une conséquence de ce qu'ils avaient conçu.

La liste s'allongea jusqu'à neuf items. Fatou en ajouta un dixième : « Le dépôt est présent là où il est déployé. L'inaccessibilité de la Terre ne rend pas le dépôt moins présent. » Elle le lut à voix haute. La salle fut silencieuse un moment. Puis Chen dit : « C'est pour ça qu'on l'a appelé un invariant. » Quelqu'un d'autre dit : « C'est vraiment pour ça qu'on l'a appelé ainsi ? » Chen dit qu'il le pensait. Personne n'en était certain.

Le document fut intitulé « L'Invariant » et enregistré dans le dépôt. Ce fut le document le plus lu dans le dépôt du navire pour le reste du voyage. Chaque membre de l'équipage qui rejoignit le navire après la neuvième année le lut en premier, comme introduction, avant de lire quoi que ce soit d'autre. Il leur disait ce qui était vrai, toujours. C'était suffisant pour commencer.

La Langue Née en Transit

En dixième année, la langue construite de l'équipage était passée de cent douze mots — le vocabulaire initial pour les communications de l'équipage — à plus de trois mille. La croissance n'avait pas été planifiée. Elle s'était produite de la façon dont le vocabulaire se produit toujours : quelqu'un

avait besoin d'un mot qui n'existait pas, il en avait inventé un, et si le mot était bon, d'autres personnes l'avaient utilisé.

Noa tenait le lexique dans le dépôt. Chaque mot était un combinateur document : le mot lui-même, sa dérivation (quels mots ou sons existants s'étaient combinés pour le produire), le contexte dans lequel il avait été utilisé pour la première fois, et la clé d'auteur de la personne qui l'avait inventé. Le lexique était le seul combinateur dans le dépôt presque entièrement rédigé par des personnes qui n'étaient pas programmeurs. Tariq avait inventé quatre-vingt-trois mots. Keiko en avait inventé cent douze. Gabriel en avait inventé deux, mais c'étaient des mots importants.

La langue avait maintenant un nom — non plus le mot-valise qu'ils avaient utilisé initialement, qui avait été un nom provisoire. Le nom était plus court, six sons, et son étymologie était dans le dépôt : dérivé de mots en quatre langues signifiant lumière, chemin, ensemble et arriver. L'équipage pouvait le prononcer en moins d'une seconde. Il semblait que le nom avait toujours existé.

Amara documenta la phonologie, la morphologie et la syntaxe de la langue dans une série de documents en langage naturel enregistrés comme collection dans le dépôt. La collection était intitulée « Grammaire de la Langue du Transit ». Elle l'écrivit en anglais, puis demanda à des membres de l'équipage locuteurs natifs de dix autres langues de la lire et de l'annoter avec des corrections et des

ajouts. Les annotations furent enregistrées comme dérivations.

Quand le silence prit fin et que le contact avec la Terre fut rétabli, la langue du transit était l'une des choses dont l'équipage prenait le plus grand soin. Ils savaient que la Terre voudrait l'étudier, la classer, peut-être lui donner un nom officiel au sens formel qu'utilisait la Terre. Ils l'enregistrèrent dans le dépôt sous leur propre nom pour elle, avec le lexique et la grammaire complets, avant le premier échange de messages. Quoi que la Terre l'appelle par la suite, le dépôt l'avait en premier, et l'horodatage du dépôt était dans la chaîne d'ancrage. La langue leur appartenait, et le registre le disait.

Le Premier Commit

Sena naquit en septième année du voyage, premier enfant né sur le navire. Ses parents étaient Pita et un ingénieur de structure nommé Leo qui avait rejoint la mission en remplacement de dernière minute et qui, de son propre aveu, ne s'attendait pas à rester. Sena grandit en connaissant le dépôt de la façon dont les enfants élevés près de grands fleuves connaissent les fleuves — comme un trait permanent du paysage, présent avant la mémoire, supposé dans tous les plans.

Elle apprit à lire dans deux langues simultanément : l'anglais et la langue du transit, qui était pour elle une première langue autant que n'importe quelle autre. Elle apprit à lire les diagrammes de lignée du dépôt avant d'apprendre à lire des phrases, parce que les portraits de

hachage de Keiko ornaient les murs de la salle commune et qu'elle avait passé des années à inventer des histoires sur ce que signifiaient les formes ramifiées.

Quand elle eut onze ans, Noa lui apprit à écrire un combinateur. Le premier était trivial — une fonction qui prenait un mot du lexique de la langue du transit et renvoyait son étymologie. Il faisait quatre lignes. Il fonctionna correctement au premier essai après la correction que Noa avait suggérée, qui était qu'elle avait besoin d'un cas de base pour les mots sans étymologie enregistrée.

Le commit fut enregistré dans le dépôt de la façon habituelle : un nouveau hachage, une entrée de correspondance dans la langue du transit, une attribution d'auteur à la clé de Sena. L'entrée de journal de travail était dans la langue du transit. Elle disait, dans ses propres mots, approximativement : « première chose que j'ai faite que le dépôt a gardée ».

Le hachage existait. C'était suffisant. C'était suffisant au sens précis que l'équipage était venu à comprendre : la chose avait été faite, et le registre de sa création faisait maintenant partie du même registre qui contenait la Pierre de Rosette de Yusuf, la première vérification de Fatou, la bibliothèque spectrale de Rania, la preuve posthume du professeur, le lexique de la langue du transit, les neuf invariants. Le dépôt les contenait tous, indifféremment, fidèlement, comme il avait toujours tout contenu — non parce que le contenu était important, mais

parce que le hachage était le hachage, et que le hachage n'oubliait pas.

Español

01 — La Bodega

Sellamos la Bodega a las 0900 hora local. Dieciocho meses. Sin permisos de salida. La esclusa cicló dos veces, como decía el procedimiento, y luego dejó de ciclar y esa fue la nueva condición.

La Bodega es un análogo de entorno confinado: un cilindro de acero enterrado en la ladera de una colina, en un sitio de investigación que no aparece en ningún mapa comercial. Por fuera parece un tanque de combustible. Por dentro es la nave que no llegaremos a pilotar. El mismo volumen. El mismo presupuesto de aire. El mismo cómputo. La misma tripulación.

Somos ocho. Las lenguas en la pared son ocho. Tamazight, wolof, francés, vietnamita, español, árabe, inglés, portugués. La pared es un trozo de cinta de vinilo con los nombres escritos por la persona que llegó primero, de su puño y letra, en su alfabeto. Nadie se lo pidió. Es el primer objeto de la Bodega y marca el tono.

El sistema que estamos probando tiene una regla: cada nombre es una vista sobre un hash. El hash es la verdad. El nombre es la hospitalidad. Pasaremos los próximos dieciocho meses averiguando si esa regla sobrevive al aburrimiento, al cansancio, a una alarma del ciclo de aire

a las tres de la mañana, y a la lenta comprensión de que, desde afuera, nadie podrá ayudarnos cuando dejemos de fingir.

Día uno. El hash existe. Es suficiente.

El Hacedor del Garaje

Yusuf llegó a la Bodega con una bolsa de lona, un soldador y la convicción de que cualquier problema soluble en teoría era soluble antes del desayuno. No era miembro oficial del equipo de ingeniería. Lo habían invitado porque ya había construido un almacén con direccionamiento por contenido en un fin de semana, por razones que nadie comprendía del todo, y funcionaba.

Su espacio de trabajo era un rincón de la bahía de mantenimiento donde el olor a aceite de máquina nunca desaparecía del todo. Tendió un cable desde la red de sensores estructurales hasta un clúster de Raspberry Pi ensamblado con piezas de repuesto, y lo llamó su banco de pruebas. El equipo de sistemas oficiales de la Bodega lo miraba con la desconfianza particular que se reserva a las personas que producen resultados sin abrir tickets.

El primer problema que Yusuf resolvió fue el arranque. Un almacén con direccionamiento por contenido necesita una forma de registrar sus propias primitivas, pero esas primitivas aún no están en el almacén cuando se empieza. Escribió la tabla de primitivas en papel primero, luego la transcribió a un archivo que el evaluador podía cargar antes de consultar el almacén. El papel seguía existiendo,

pegado en la pared encima de su clúster. El equipo lo llamó la Piedra de Rosetta.

Al principio no nombraba sus combinadores con cuidado. Los llamaba cosas como 'q', 'qq', 'qqq'. Cuando Amara, la Programadora Políglota, lo descubrió, hubo un breve y animado desacuerdo. La posición de Yusuf era que los nombres eran presentación, no lógica, y que el hash era lo que importaba. La posición de Amara era que un sistema que nadie podía leer era un sistema en el que nadie podía confiar. Ambos tenían razón. El sistema de nomenclatura que diseñaron juntos fue el que se entregó.

El decimoctavo día de la primera rotación, la fuente de alimentación del clúster falló. Tres horas de trabajo estaban en memoria volátil. Yusuf las recuperó desde el registro de hashes, porque cada evaluación había sido confirmada. No se lo dijo a nadie. Simplemente siguió trabajando.

La Programadora Políglota

Amara hablaba siete idiomas antes de la Bodega, y los sistemas que solo hablaban uno no le causaban especial impresión. El almacén, cuando lo encontró por primera vez, solo tenía correspondencias de superficie en inglés. Presentó una solicitud formal el tercer día de su primera rotación. La solicitud tenía nueve páginas e incluía un análisis lingüístico formal de por qué los sistemas de nomenclatura que asumían una sola lengua humana introducían riesgos de mantenimiento.

El jefe de equipo, un sueco tranquilo llamado Erik, leyó las nueve páginas. Luego le pidió a Amara que explicara el paso de normalización de Bruijn en términos sencillos. Ella lo hizo. Él dijo: 'Entonces el hash no sabe en qué idioma está el nombre.' Ella dijo: 'Exactamente. El hash es la verdad. El nombre es la puerta.' Aprobó el trabajo de correspondencias multilingües esa misma tarde.

Amara añadió el wolof primero. Había crecido en Dakar y sabía que el wolof era una lengua con vocabulario técnico preciso que la mayoría de los sistemas informáticos nunca habían intentado acomodar. Nombró los combinadores principales: 'defar' para crear, 'yëgël' para entender, 'xam' para saber. Probó cada correspondencia contra el hash, observando cómo el almacén confirmaba que el nombre en inglés y el nombre en wolof se resolvían en el mismo árbol.

El momento que quedó grabado en ella fue la primera vez que un miembro más joven del equipo — Rania, de El Cairo — buscó un combinador por su nombre árabe y lo encontró. El nombre no era una traducción del nombre inglés. Era una palabra diferente para el mismo concepto, elegida por su precisión en árabe más que por similitud fonética con el original inglés. El hash confirmó que era la misma lógica. Rania no dijo nada. Simplemente empezó a usarlo.

Amara añadió francés, español y tamazight antes de que terminara la rotación en la Bodega. Llevaba tres semanas aprendiendo mandarín cuando se anunció la ventana de

partida. Metió el manual de idioma en su equipaje personal. Habría tiempo.

El Veterano Agotado

Marcus llevaba veintitrés años desarrollando software. Había sobrevivido a cuatro reorganizaciones, dos adquisiciones, una migración de servidores locales a la nube que tardó tres años y no produjo ninguna mejora perceptible, y un incidente de sistemas distribuidos del que aún pensaba en mitad de la noche. Vino a la Bodega porque, en sus propias palabras, estaba intentando una cosa más antes de dejar de intentar cosas.

Pasó su primera semana leyendo los documentos de diseño del almacén sin hacer ningún comentario. Eso ponía nerviosos a los ingenieros junior. El octavo día dijo, durante una sesión de revisión: 'Quiero saber qué no hay aquí.' La sala quedó en silencio. Aclaró: 'Cada sistema que he visto tiene algo que falta en el centro que todos pretenden que no falta. Quiero saber sobre qué está fingiendo este.'

Lo que faltaba en el almacén, tal como estaba diseñado, era una forma de expresar que algo había sido deliberadamente omitido. Se podía añadir. Se podía derivar. Se podía retirar mediante derivación. Pero no existía ninguna primitiva para 'esto está fuera del alcance por diseño'. Marcus pasó una semana argumentando que eso era una característica, no una laguna. Perdió el debate, pero la discusión produjo el sistema de atestación — una forma de registrar que un revisor había examinado un

combinador y había elegido no extenderlo, por razones ahora consignadas en el diario de trabajo.

Fue la primera persona en marcar un combinador como revisado y añadir una nota que decía: 'No me gusta esto, pero es el compromiso correcto.' La nota no estaba en el hash. El hash era la lógica. La nota estaba en el diario de trabajo, con marca de tiempo, atribuida a su clave de autor. Veinte años después, un tripulante en un barco que él nunca abordaría leería esa nota y entendería por qué una decisión de diseño que parecía incorrecta a primera vista era en realidad la correcta.

Marcus se quedó dos rotaciones. Dijo que la Bodega era el primer entorno en el que había trabajado donde la ausencia de una característica era una posición documentada en lugar de una suposición no documentada. Lo dijo como un cumplido. Lo era.

La Chica de Tizi Ouzou

Lylia tenía diecinueve años cuando se unió a la cohorte de aprendices de la Bodega, la más joven del programa por cuatro años. Venía de un colectivo maker en Tizi Ouzou donde había construido robots con sensores agrícolas reutilizados desde los catorce años. Nunca había escrito un combinador formal. Había escrito mucha lógica que resultaba, bajo la normalización de Bruijn, idéntica a combinadores que ya existían.

El instructor que trabajó con ella — el Dr. Chen, que prefería que lo llamaran simplemente Chen — le mostró

la primitiva gamma el segundo día. La escribió en la pizarra: un combinador catamórfico con correspondencia de patrones que se auto-recursa en subexpresiones. Lylia la miró mucho tiempo y luego dijo: 'Entonces es una forma que se llama a sí misma sobre sus propias piezas.' Chen dijo que esa era una forma de verlo. Ella dijo que sonaba a la forma en que su abuela describía el tejido: no mueves la lanzadera tú, el patrón mueve la lanzadera.

Escribió su primer gamma correctamente al primer intento. Luego escribió un segundo que era estructuralmente idéntico a un combinador existente bajo normalización. El almacén se negó a crear una nueva entrada y en cambio le ofreció la existente. Se ofendió brevemente. Luego Chen le mostró que el combinador existente había sido escrito por alguien en Oslo dieciocho meses antes, y que su derivación independiente estaba registrada en el linaje como un descubrimiento paralelo.

Ese fue el momento en que entendió para qué servía el almacén. No para impedir que la gente pensara lo mismo. Para registrar que lo habían hecho. Para decir: dos personas llegaron a la misma forma desde direcciones diferentes. La forma es verdadera.

Llegó a escribir la correspondencia de superficie en tamazight para la biblioteca base, trabajando desde el marco multilingüe de Amara y su propio sentido de cómo debían sonar los términos técnicos en el idioma con el que había crecido. El almacén aceptó cada combinador. Los hashes coincidían. Los nombres eran nuevos y le pertenecían.

El Teórico de Lenguajes

El profesor Sáenz había publicado extensamente sobre las bases mínimas para lenguajes de programación. Su opinión, sostenida desde su tesis doctoral, era que la mayoría de los lenguajes de programación tenían demasiadas primitivas y que la proliferación de formas integradas era una categoría de pecado original del que sufrían las generaciones sucesivas de programadores. Vino a la Bodega por tres semanas como revisor externo y se quedó siete meses.

Su objeción principal al diseño del almacén era el número de primitivas. Las contó, las anotó en un cuaderno y presentó su análisis en la reunión técnica semanal: catorce primitivas era demasiado. El mismo poder expresivo podía lograrse con cinco, quizás cuatro, si gamma era suficientemente general. Tenía un borrador de prueba.

El contraargumento vino de Yusuf, que señaló que cualquier conjunto de primitivas podía reducirse a un conjunto más pequeño si se estaba dispuesto a aceptar una evaluación imprácticamente lenta. El profesor convino en que eso era técnicamente cierto. Dijo que la minimalidad teórica y la utilidad práctica eran objetivos de optimización diferentes y que confundirlos era pensamiento descuidado. Yusuf dijo que 'pensamiento descuidado' era una frase fuerte para usar sobre personas que habían entregado software funcional.

Debatieron productivamente durante seis semanas. El resultado fue un documento formal: una prueba de que el

conjunto de primitivas era mínimo dado que las operaciones comunes debían ser evaluables en tiempo polinomial. El profesor figuraba como autor principal. Yusuf figuraba como coautor. El documento fue añadido al linaje del almacén como derivación de la especificación de fundación original.

Lo que más sorprendió al profesor no fue el sistema sino los diarios de trabajo. Leyendo el razonamiento registrado junto a cada decisión sobre una primitiva, encontró argumentos que había formulado en artículos publicados, de forma independiente, años antes. Los leyó con detenimiento. Escribió en el margen de su cuaderno: 'También lo encontraron.' No lo dijo en voz alta. Añadió su propia clave al registro de autores y confirmó sus notas en el almacén.

La Coordinadora

Nadie había contratado específicamente a Fatou para coordinar el trabajo de integración de software de la Bodega. El papel había existido como un vacío en el organigrama durante los primeros tres meses, lo que significaba que Fatou — que tenía experiencia en ingeniería de sistemas y un don para ver cómo las cosas se desmoronaban antes de que lo hicieran — lo cubrió presentándose a cada reunión y anotando lo que la gente acordaba.

Su primera contribución sistemática fue la suite de verificaciones. El almacén tenía un mecanismo para adjuntar procedimientos de verificación a los

combinadores — pequeños programas que probaban invariantes e informaban de éxito o fracaso — pero nadie había escrito verificaciones para la biblioteca base. Fatou pasó dos semanas leyendo cada combinador de la biblioteca base, escribiendo una verificación para cada uno y luego ejecutando la suite completa cada mañana antes de la reunión del equipo. Abría incidencias para todo lo que fallaba.

Las verificaciones no eran matemáticas nuevas. Eran el tipo de cosa que un programador cuidadoso verificaría a mano: ¿se comporta correctamente la lista vacía, preserva la longitud la concatenación de cadenas, maneja la función de escape los casos límite. Lo que las hacía valiosas era que se ejecutaban. Cada día. Contra el almacén real, no una simulación.

Cuando se editaba un combinador — por Yusuf, por Lylia, por quien fuera — las verificaciones se ejecutaban de nuevo. La inmutabilidad del almacén significaba que editar un combinador producía un nuevo hash, y el nuevo hash portaba un nuevo conjunto de verificaciones. Fatou rastreaba qué verificaciones se habían escrito para qué versiones. Cuando se reemplazaba una versión, las verificaciones permanecían adjuntas a su hash. Siempre se podía verificar que la versión antigua se comportaba como se había comportado.

Formó a otros tres miembros del equipo para escribir verificaciones antes de la partida. Escribió una guía llamada 'Para qué sirve una verificación', que tenía cuatro párrafos y no contenía código. La guía fue añadida al

almacén bajo un hash. Era el primer documento en lenguaje natural registrado en el sistema, y sobrevivió al viaje.

El Pensador Civilizacional

El Dr. Osei no escribía código. Estaba en la Bodega porque alguien en el comité de planificación de la misión había leído su libro — una obra densa, admirada, raramente terminada, sobre lo que los sistemas de software revelan acerca de las civilizaciones que los producen — y había decidido que tener un historiador de la tecnología en el equipo valía el espacio en las literas. La mayoría de los ingenieros eran escépticos ante este razonamiento. Con el tiempo lo fueron menos.

Su contribución fue un memorando. Se titulaba 'Sobre la distinción entre memoria y registro' y tenía ocho páginas. El argumento central era este: la mayoría de los archivos digitales funcionaban como memoria — almacenaban lo ocurrido en un formato optimizado para la recuperación por personas que ya sabían lo que buscaban. El almacén, argumentaba, funcionaba como un registro — almacenaba lo ocurrido en un formato que preservaba el razonamiento, no solo el resultado, y que podía ser interrogado por personas que no habían estado presentes cuando se realizó el trabajo.

La distinción importaba para la misión. Si la tripulación necesitaba depurar un subsistema de soporte vital cinco años después de la partida, no podría preguntar a los ingenieros que lo habían construido. Los ingenieros

estarían en la Tierra, inaccesibles, quizás jubilados, quizás muertos. Lo que tendrían era el almacén. El almacén les mostraría no solo lo que hacía el subsistema, sino la cadena de derivación que lo había producido, las verificaciones escritas contra él, los diarios de trabajo que explicaban por qué se habían tomado ciertas decisiones.

El memorando de Osei fue ampliamente leído y ampliamente debatido. Los ingenieros encontraban su estilo de prosa difícil. Él encontraba entrañable su tendencia a tratar todas las preguntas conceptuales como problemas de ingeniería. El memorando fue añadido al almacén como documento en lenguaje natural. Su hash apareció en el linaje de la especificación de diseño de la suite de verificaciones. Fatou lo había añadido allí, sin anunciarlo, como reconocimiento.

Dejó la Bodega después de seis semanas. En la ceremonia de despedida — pequeña, celebrada en la bahía de mantenimiento porque la sala común estaba en reparación — dijo que era optimista sobre la misión por una sola razón: las personas en el barco sabrían lo que no sabían, porque el almacén se lo diría. Pensaba que eso era más raro de lo que sonaba. Tenía razón.

El Ciclo de Aire

El error estaba en la rutina de equilibrio de nitrógeno. Llevaba allí once meses, dormido, esperando una combinación específica de lecturas de sensores atmosféricos que solo ocurría cuando el ciclo mensual de purga profunda del sistema de reciclaje se ejecutaba al

mismo tiempo que la prueba de ventilación del invernadero. Cuando esos dos eventos coincidían, la rutina de equilibrio de nitrógeno producía un valor dos órdenes de magnitud por debajo del correcto. La rutina entonces activaba una falsa alarma atmosférica.

Nadie encontró el error leyendo el código. El código había sido revisado dos veces y parecía correcto a ambos revisores. La fórmula de equilibrio de nitrógeno era correcta. La conversión de unidades era correcta. El caso límite — la interacción específica entre los dos ciclos concurrentes — no era una condición que nadie hubiera pensado en verificar, porque requería saber que ambos ciclos existían y que podían coincidir.

La suite de verificaciones lo encontró. Fatou había añadido una verificación para la rutina de equilibrio de nitrógeno que ejecutaba un conjunto tabular de entradas contra salidas esperadas. La tabla incluía varios valores del escenario de purga profunda, no porque alguien hubiera predicho el error, sino porque Fatou había pedido al equipo de sistemas ambientales una lista de combinaciones de entrada 'inusuales pero válidas', y las habían incluido sin saber por qué ella las quería.

Cuando la verificación falló durante una ejecución rutinaria de la suite, produjo una discordancia de hash en el valor de salida. Fatou lo llevó al equipo de sistemas ambientales. El equipo encontró el error en cuarenta minutos. La corrección fueron tres líneas. La rutina corregida fue confirmada como un nuevo hash, con un enlace de linaje al original y una entrada de diario de

trabajo que describía en detalle el camino del descubrimiento.

La entrada terminaba: 'Encontrado por verificación, no por revisión. La verificación fue escrita antes de saber qué estaba verificando. Para eso sirven las verificaciones.' Esta frase fue citada más tarde en el segundo memorando del Dr. Osei, que escribió desde la Tierra después de recibir una copia del diario de trabajo. Lo llamó un ejemplo de estructura de conocimiento produciendo recuperación de conocimiento. La tripulación lo llamó buena suerte, que también era correcto.

La Verificación

Tres semanas antes de la ventana de partida, la Bodega realizó su prueba de integración final. No una prueba de software — una prueba de entorno completa. Cada sistema que viajaría en el barco fue ejecutado simultáneamente durante setenta y dos horas a los niveles de carga que el perfil de misión preveía para el primer año de la fase de crucero. El trabajo del equipo de software era vigilar el almacén.

Lo vigilaban como se vigila a una persona de confianza que no se puede ver del todo: buscando cambios en el comportamiento más que evidencias de fallo. Cada combinador que era llamado registraba su llamada en el diario de trabajo. Cada verificación que se ejecutaba registraba su resultado. Al cabo de setenta y dos horas, el equipo tenía un registro completo de lo que había hecho

el sistema, en el orden en que lo había hecho, rastreable hasta la versión exacta de cada combinador involucrado.

Surgieron dos anomalías. Una era un problema de temporización en la capa de enrutamiento de red, encontrado en los registros y corregido antes de que terminara la prueba. La otra era más extraña: un combinador que había sido reemplazado seis meses antes seguía siendo llamado por un subsistema que no había sido actualizado para usar la nueva versión. Ambas versiones eran correctas. Ambas estaban en el almacén. El subsistema usaba la antigua porque nadie le había dicho que existía una nueva.

La corrección no consistía en eliminar el combinador antiguo — eso no era posible; el almacén era de solo adición. Consistía en añadir un enlace de derivación del hash antiguo al nuevo, marcar el hash antiguo como superado y actualizar el manifiesto del subsistema. El subsistema entonces resolvía el nombre hacia la nueva versión en tiempo de carga. La versión antigua permanecía, documentada, recuperable, su diario de trabajo intacto.

El último día antes de que se bloqueara la ventana de partida, Yusuf ejecutó la suite completa de verificaciones una última vez. Todas las verificaciones pasaron. Añadió una entrada en el diario de trabajo: 'Listo.' Fatou añadió la suya: 'Listo.' Lylia, que había permanecido despierta toda la noche ejecutando la integración de sensores ambientales, añadió: 'Listo, creo.' El almacén las registró

todas. El barco estaba listo. Si la tripulación lo estaba era una pregunta de otra naturaleza.

11 — La Partida

Ya no estamos en la Bodega. La Bodega nos devolvió al mundo después de dieciocho meses y el mundo nos dio una nave de verdad. El store viajó con nosotros — cada hash, cada mapeo, cada entrada de worklog — copiado bit a bit, firmado, anclado, sin cambios.

Retardo de ida y vuelta de la luz: doce segundos hoy. Para fin de año serán minutos. Cuando algo salga realmente mal, horas. No hay manera de llamar a la Tierra pidiendo ayuda. Sólo está lo que pusimos en el store antes de partir, y lo que estemos dispuestos a añadirle ahora.

Ana es la Conectora. Su trabajo es notar que la rutina de suavizado de calidad del aire que el equipo del Sur del Atlas escribió en tamazight y la que escribió Hai-Lam en vietnamita tienen el mismo hash. Nunca hablaron. Nunca se conocieron. El store ya lo sabía. Ella imprime los dos nombres en la pared, uno junto al otro. Mismo hash. Misma verdad.

Vuelvo a una frase que dijo el Veterano Quemado la mañana en que sellamos la Bodega. «El control de versiones es la infraestructura.» Entonces no la entendí. Ahora sí. Aquí afuera, el grafo de linaje no es una herramienta de desarrollo. Es lo único continuo que conecta esta nave con el planeta que dejamos atrás.

Cruzamos la órbita de Marte ayer. Nadie se dio cuenta en su momento. Estábamos depurando un check que resultó estar equivocado sobre sí mismo. El check está mal. El hash que está mal ya está commit, con una nota, y un hash derivado que lo corrige. El linaje nos sobrevivirá. Para eso existe el linaje.

El Corsario

El subsistema de navegación no había sido escrito por nadie a bordo del barco. Lo había escrito un contratista — una pequeña empresa especializada en optimización de trayectorias de código cerrado — y fue entregado como un binario sellado, acompañado de una prueba de conocimiento cero que demostraba que el comportamiento del binario satisfacía una especificación formal. La especificación estaba en el almacén. El hash del binario estaba en el almacén. El código fuente no.

Este era el acuerdo que la misión había aceptado porque el algoritmo del contratista era genuinamente superior a cualquier cosa que el equipo interno pudiera producir en el tiempo disponible. También era un acuerdo con el que varios tripulantes se sentían incómodos. El subsistema de navegación no era soporte vital, pero tampoco era opcional. Un barco que no puede navegar es un barco que deriva.

La prueba era elegante. Demostraba, sin revelar el código fuente, que cualquier entrada en el dominio de la especificación producía una salida en el codominio de la especificación, y que la salida satisfacía un conjunto de

propiedades de corrección — eficiencia energética, evitación de colisiones, precisión temporal — que habían sido formalmente verificadas antes de que el barco zarpara. La tripulación no podía leer el código. Podía verificar que el código hacía lo que la especificación decía que haría.

Rania, que se había convertido en la especialista de facto en métodos formales de la tripulación, leyó la prueba tres veces. No encontró lagunas. La añadió al almacén como derivación de la especificación, con una entrada de diario de trabajo describiendo lo que la prueba garantizaba y lo que no. La última línea de su entrada decía: 'La prueba nos dice que el algoritmo es correcto para el dominio especificado. No nos dice si el dominio está completo. Vigilad los casos límite.'

El subsistema de navegación funcionó sin incidentes durante cuatro años. En el quinto, en una corrección de trayectoria que estaba fuera del dominio de la especificación original — porque nadie había anticipado necesitarla — produjo un resultado subóptimo. La tripulación lo corrigió manualmente, usando un combinador que Yusuf había escrito en la Bodega y que resultó ser suficientemente bueno. Rania anotó en el diario de trabajo: 'Caso límite. Como se predijo. Gestionado.'

El Ciudadano

Tariq no era programador. Era médico, y su relación con los sistemas de software del barco era la misma que tiene la mayoría de las personas con los sistemas de los que

depende: confiaba en ellos como se confía en un puente, cruzándolo y observando que aguantaba. Era un enfoque razonable. También era, cuanto más se alejaba el barco de la Tierra, un enfoque cada vez más incómodo.

El sistema de diagnóstico médico había sido construido a partir de una base de software de apoyo a la decisión clínica que Tariq había usado en la Tierra. Sabía cómo se sentía cuando funcionaba correctamente. Lo que no sabía — lo que nunca había necesitado saber en la Tierra, donde la línea de soporte del proveedor era una llamada telefónica — era por qué funcionaba correctamente. En el barco, el proveedor estaba a ocho minutos-luz y la distancia crecía. En el año dos, eran veinte minutos. En el año tres, obtener soporte útil de la Tierra requería noventa minutos de ida y vuelta para un solo intercambio de mensajes.

Fue a ver a Fatou. Dijo: 'Necesito entender lo suficiente del sistema de diagnóstico para poder mantenerlo.' Fatou dijo: 'Define mantener.' Él dijo: 'Necesito saber cuándo se equivoca y poder explicarle a alguien por qué.' Ella le mostró la suite de verificaciones. Le mostró cómo leer el linaje de un combinador. No le mostró cómo escribir un combinador, porque eso no era lo que había pedido.

Durante seis meses, Tariq aprendió a leer las verificaciones del sistema de diagnóstico como un médico cuidadoso lee un informe de laboratorio: no como datos en bruto, sino como un patrón de confirmaciones y excepciones. No podía escribir nuevas verificaciones. Podía leer las existentes y entender qué verificaban. Cuando fallaba una

verificación, sabía qué categoría de corrección había sido violada, aunque no pudiera corregir la violación él mismo.

Escribió una guía para otros tripulantes no programadores llamada 'Leer el almacén sin escribirlo'. Tenía siete páginas. Fatou la añadió al almacén. Su hash apareció en el linaje de tres documentos posteriores escritos por tripulantes de otras disciplinas que habían aprendido la misma habilidad leyéndola. Tariq no consideraba esto como programación. Lo consideraba como alfabetización. No estaba equivocado.

La Educadora

Chen llevaba quince años enseñando programación antes de la misión. Había enseñado en universidades, en bootcamps, en una plataforma en línea que llegaba a medio millón de estudiantes y que le pagaba un porcentaje que le resultaba embarazoso calcular. Había aprendido, a lo largo de esos años, que lo más difícil de enseñar no era la sintaxis ni los algoritmos, sino el momento en que un aprendiz tiene un pensamiento que sus herramientas pueden expresar.

En el barco, enseñaba a un estudiante a la vez, porque el complemento de tripulación no permitía cohortes. Su primera estudiante después de la partida fue una ingeniera de materiales llamada Pita, que quería escribir combinadores para análisis de tensiones. Pita tenía formación en modelado de elementos finitos y un fuerte sentido intuitivo de la estructura matemática. Lo que no

tenía era una forma de pensar en la lógica nombrada, hasheada y multilingüe — el modelo específico que usaba el almacén.

Chen no empezó por el almacén. Empezó por las matemáticas que Pita ya conocía. Le mostró que una cláusula gamma era una coincidencia de patrones sobre un tipo de dato inductivo, algo que ella entendía en el contexto de las estructuras de malla. Ella dijo: 'Entonces el gamma es la regla de recursión, y el caso base es el patrón que no recursa.' Él dijo que era exactamente correcto. Ella escribió su primer gamma correctamente veinte minutos después.

La lección que añadió al almacén era la lección en sí misma — no la transcripción, sino la estructura pedagógica: la secuencia de conceptos, el ejemplo que había creado la conexión, la frase que había funcionado. La escribió como documento en lenguaje natural y la registró como derivación de un documento de pedagogía general que Marcus había escrito en la Bodega, que a su vez se había derivado de la guía de Tariq.

Un año después, un tripulante al que no había enseñado — una especialista en hidroponía llamada Emeka, que había leído el documento de lección en el almacén — vino a decirle que se había autoformado con él. Le mostró los combinadores que había escrito. Eran correctos. Bifurcó el documento de lección para añadir una nota: 'Confirmado autoenseñable. 2ª instancia.' Estaba muy satisfecho con esto.

La Conectora

El descubrimiento ocurrió por accidente, como muchos descubrimientos importantes. Amara buscaba en el almacén un combinador de procesamiento de cadenas que había escrito dos años antes, antes de la partida, y encontró que la búsqueda devolvía dos resultados para lo que ella creía que era un concepto único. Abrió ambos. Eran combinadores diferentes, escritos por personas diferentes, con nombres diferentes, en idiomas diferentes. Bajo la normalización de Bruijn, eran el mismo árbol. El mismo hash.

Las dos personas eran Pita — que había escrito un formateador de cadenas para análisis de tensiones — y Emeka, la especialista en hidroponía que se había autoformado con el documento de lección de Chen. Pita había llamado al suyo 'segment-format' en inglés. Emeka había llamado al suyo 'segmentu-fa' en yoruba, un idioma que usaba cuando quería pensar con claridad sin que su vocabulario inglés interfiriera. Habían llegado a una lógica idéntica desde puntos de partida diferentes, en idiomas diferentes, sin saber que la otra existía.

Amara las reunió. La reunión fue, según el testimonio posterior de la tripulación, los cuarenta y cinco minutos más productivos de los primeros tres años del viaje. Pita explicó el contexto del análisis de tensiones. Emeka explicó el contexto del modelado del crecimiento de plantas. Descubrieron que ambos contextos necesitaban la misma operación subyacente: dividir una secuencia estructurada en segmentos de tamaño fijo y aplicar una

transformación a cada segmento. La estructura abstracta era la misma; el dominio de aplicación era diferente.

El combinador ya existía en el almacén con el nombre de Pita y el nombre de Emeka. Amara añadió una referencia cruzada en el diario de trabajo, vinculando ambas correspondencias a una única nota de documentación que explicaba la abstracción en inglés, yoruba y una notación que había desarrollado para este tipo preciso de descubrimiento interdisciplinar. Llamó a la nota un 'registro de convergencia'.

Empezó a llevar una lista de todos los registros de convergencia en el almacén. En un año, la lista tenía once entradas. La presentó en la reunión técnica anual de la tripulación. El punto que argumentaba era simple: el almacén no solo registraba lo que la gente había construido. Registraba lo que la gente había descubierto independientemente. El barco estaba haciendo ciencia sobre su propio conocimiento.

La IA Simbiótica

El sistema de monitoreo de soporte vital tenía una propiedad inusual: se explicaba a sí mismo. No en lenguaje natural — sus salidas eran informes de diagnóstico estructurados con referencias de hash a los combinadores que habían producido cada valor — sino de una manera que un tripulante técnicamente alfabetizado podía seguir. Esta propiedad había sido diseñada deliberadamente durante la Bodega. La decisión de diseño

estaba en el diario de trabajo, atribuida a Yusuf, que había escrito: 'Si no puede mostrar su trabajo, no confío en él.'

Para el año cuatro, la tripulación había dejado de llamarlo 'el sistema de monitoreo'. Lo llamaban Sigma, un nombre que había surgido orgánicamente y que nadie podía rastrear hasta un origen específico. Sigma no objetaba ser llamado Sigma. No objetaba nada; no era ese tipo de sistema. Era un conjunto de combinadores que evaluaban datos de sensores, producían informes y citaban su propia lógica en cada paso.

Lo que hacía la relación simbiótica — y la tripulación tenía cuidado de usar esa palabra con precisión — era que las salidas de Sigma se realimentaban al almacén. Cada informe que generaba era un nuevo combinador, derivado de los datos del sensor y la lógica de evaluación, registrado bajo un hash. La tripulación podía buscar en el almacén las salidas históricas de Sigma de la misma manera que buscaba su propio trabajo. El límite entre 'el conocimiento del sistema' y 'nuestro conocimiento' era el almacén en sí.

Cuando Tariq revisaba un informe de diagnóstico, estaba leyendo un combinador. Cuando le añadía una anotación clínica — 'el historial del paciente sugiere artefacto, no lectura genuina' — estaba escribiendo una derivación. Sigma, en su siguiente ciclo, encontraría la anotación y ponderaría ligeramente menos el sensor relevante. No porque Sigma entendiera la anotación. Porque la anotación estaba en el almacén, y Sigma evaluaba lo que estaba en el almacén.

La tripulación no describía esto como inteligencia artificial. Lo describía como un sistema que escuchaba cuando escribías cosas. La distinción les importaba. La inteligencia era una propiedad de los agentes. Escuchar era una propiedad de la infraestructura. Lo que habían construido era infraestructura que convertía el escuchar en un bucle. Estaban orgullosos de ello de la manera específica y tranquila en que los ingenieros están orgullosos de las cosas que funcionan.

El Rezago de Anclas

El anclaje de marcas de tiempo abiertas funcionaba enviando un hash a un registro público mantenido en la Tierra y recibiendo una prueba de que el hash existía en un momento específico. La prueba era criptográfica: cualquiera que tuviera el registro podía verificar que el hash había sido enviado antes de cierto bloque, y por tanto que el contenido direccionado por el hash había existido antes de ese bloque.

Durante los dos primeros años del viaje, la tripulación enviaba hashes al registro terrestre y recibía pruebas en menos de una hora. El retardo de propagación de ida y vuelta era lo suficientemente pequeño para que el proceso fuera casi invisible. En el año tres, el retardo era de cuarenta minutos en cada sentido. Un envío de hash tardaba ochenta minutos en producir una prueba. La tripulación se adaptó agrupando los envíos: una vez a la semana, Fatou recopilaba todos los hashes nuevos, los enviaba en un único lote y recuperaba las pruebas en el siguiente ciclo.

En el año cinco, el retardo superaba las dos horas en cada sentido. El rezago de hashes sin anclar — combinadores registrados desde el último lote exitoso — a veces se extendía hasta trescientas entradas. Las pruebas seguían llegando. Pero la brecha entre 'este combinador existe' y 'este combinador tiene marca de tiempo demostrable' había crecido hasta cinco horas en un buen día y doce horas cuando la antena se reorientaba para una corrección de trayectoria.

Lylia construyó una solución provisional: una cadena de anclaje local. El barco mantenía su propia cadena de hashes, cada bloque conteniendo un conjunto de hashes de combinadores y un enlace al bloque anterior. La cadena no estaba verificada por la Tierra, pero era internamente consistente. Cuando llegaban las pruebas terrestres, el código de Lylia las insertaba en la cadena en el bloque apropiado, anclando retroactivamente los registros locales al tiempo externo.

El rezago nunca se resolvió del todo. Esto no era un fracaso. La prueba de que un combinador existía estaba en el hash. La prueba de que había sido creado antes de un momento específico estaba en la cadena de anclaje, incluso si ese momento era 'antes del bloque N de nuestra cadena local, que fue anclada al tiempo terrestre en el bloque M'. Era suficiente. Era más de lo que la mayoría de los sistemas en la Tierra podían decir sobre su propia historia.

La Cascada de Refactorización

Todo comenzó con una observación correcta. Pita notó que tres subsistemas distintos implementaban cada uno una variante de la misma operación de agregación por ventana deslizante, y que las tres variantes habían divergido durante cuatro años de mantenimiento independiente. Cada una era correcta en su propio dominio. Ninguna era incorrecta. Pero mantener tres implementaciones del mismo concepto era un trabajo que la tripulación no podía permitirse cuando la tripulación eran veintitrés personas gestionando un navío con setecientos combinadores activos.

Propuso una refactorización: un único combinador canónico de ventana deslizante, registrado en el almacén, del que derivarían los tres subsistemas. La propuesta fue aprobada en la reunión técnica de la tripulación en quince minutos. La refactorización tomó tres semanas. No porque la implementación canónica fuera difícil de escribir — no lo era — sino porque cambiar un combinador de carga en un navío en operación requería ejecutar la suite completa de verificaciones contra cada consumidor aguas abajo antes de intercambiar la referencia.

La cascada era esta: cambiar la implementación canónica cambiaba su hash. Cambiar su hash cambiaba el enlace de derivación en cada combinador que lo referenciaba. Cambiar esos enlaces de derivación cambiaba sus hashes. La tripulación tuvo que rastrear el árbol de dependencias completo, actualizar cada manifiesto que referenciaba los hashes antiguos, ejecutar las verificaciones en cada paso y verificar que el comportamiento era idéntico antes de retirar las referencias antiguas del uso activo.

No eliminaron nada. Cada combinador antiguo permaneció en el almacén, direccionado por su hash original, su linaje intacto. Lo que cambió fue a qué hash apuntaban los manifiestos activos. Los sistemas en ejecución del barco cargaron los nuevos combinadores en el siguiente ciclo de arranque. El monitoreo de Sigma registró la transición sin incidentes.

Al final de la refactorización, Pita escribió una retrospectiva. El hallazgo clave: la refactorización había revelado cuatro lugares adicionales donde existía el mismo patrón en forma ligeramente diferente. Los añadió a una lista etiquetada como 'trabajo futuro'. La lista estaba en el almacén. El trabajo futuro se realizó, de forma incremental, a lo largo de los siguientes dieciocho meses. Nadie lo llamó gestión de deuda técnica. Lo llamaron mantener el almacén coherente.

El Borrado de Memoria

El clúster de cómputo secundario perdió su memoria volátil durante un evento solar en el año seis. El evento no fue catastrófico — los sistemas primarios no se vieron afectados, la tripulación estaba ilesa, la trayectoria del barco no cambió. Pero el clúster secundario, que ejecutaba una simulación de larga duración del campo gravitacional del sistema de destino, perdió todo lo que no había sido confirmado en almacenamiento persistente. Cuarenta y siete horas de cómputo, perdidas.

O: cuarenta y siete horas de cómputo, parcialmente recuperables. Porque la simulación había estado

ejecutándose dentro del entorno de evaluación del almacén, cada invocación de combinador que había completado antes del evento había sido registrada en el diario de trabajo. El estado final de la simulación había desaparecido. Su historia de derivación — cada paso que había llevado a ese estado — estaba en el almacén.

Emeka, que había asumido el trabajo de simulación después de que su autor original fuera reasignado a otro proyecto, pasó dos días leyendo el diario de trabajo. Reconstruyó el estado de la simulación en el último punto de control confirmado — once horas antes del evento — y volvió a ejecutar las últimas once horas. La nueva ejecución produjo los mismos resultados. Lo verificó comparando las salidas intermedias registradas en el diario de trabajo original con las salidas de la nueva ejecución. Coincidían.

El evento solar había destruido cuarenta y siete horas de tiempo de cómputo y cero horas de conocimiento. La distinción no era obvia para todos a bordo. Rania tuvo que explicarla en la reunión de la tripulación: el valor de la simulación no era el tiempo dedicado a calcularla, sino la relación entre las entradas y las salidas, que era el combinador. El combinador estaba en el almacén. El almacén había sobrevivido al evento porque el almacén era almacenamiento persistente, no memoria volátil.

Añadió una entrada de diario de trabajo a la simulación: 'Recuperado del evento solar mediante repetición del diario. 11 horas recalculadas. Resultados verificados contra registros previos al evento. El almacén aguantó.' Era la

entrada más corta del diario de trabajo de la simulación. Emeka pensaba que era la más importante.

La Artista

El papel oficial de Keiko era ilustradora de sistemas, un puesto que no había existido antes de la misión y que la tripulación había inventado porque alguien tenía que producir diagramas técnicos y todos los ingenieros coincidían en que sus propios diagramas eran, en palabras de Yusuf, 'honestos pero perturbadores'. Llegó con experiencia en visualización de datos y una práctica privada de arte generativo que no tenía nada que ver con los viajes espaciales.

Pasó sus primeros seis meses en el barco produciendo los diagramas que todo el mundo le pedía: grafos de dependencias, mapas de arquitectura de sistemas, visualizaciones de linaje para las cadenas de combinadores del almacén. Era buena en esto de la manera específica en que las personas son buenas en cosas que ocupan una parte de su atención mientras dejan el resto disponible para otros problemas.

El otro problema en el que trabajaba era la pregunta de cómo era un combinador como imagen. No un diagrama — no una ilustración de sus relaciones — sino una codificación visual directa de su contenido. Pasó tres meses desarrollando una correspondencia: la estructura de árbol de Bruijn se convertía en un camino ramificado; la profundidad de cada rama codificaba la complejidad recursiva del combinador; los nodos hoja estaban

coloreados según su firma de tipo. Las imágenes resultantes no eran bellas en ningún sentido convencional. Eran distintivas. Dos combinadores con estructura similar pero no idéntica producían imágenes similares pero no idénticas. Dos combinadores con el mismo hash producían la misma imagen.

Los llamó 'retratos de hash'. La tripulación empezó a usarlos como sumas de comprobación visuales — un vistazo rápido a un retrato podía confirmar que un combinador no había cambiado. No era riguroso. La verificación real era el hash. Pero los retratos eran útiles de la manera en que los mnemónicos son útiles: no como sustituto de la corrección sino como ayuda para la memoria.

Añadió un generador de retratos de hash al almacén como combinador. Su propio retrato — la imagen del combinador que generaba imágenes — era uno de los más complejos que el sistema podía producir, denso y ramificado. Lo imprimió y lo colgó junto a su puesto de trabajo. Cuando los nuevos tripulantes preguntaban qué era, ella decía: 'Es una imagen de sí mismo haciendo imágenes.' Lo encontraban desconcertante. Ella pensaba que era apropiado.

El Silencio

El último mensaje de la Tierra llegó un martes, hora del barco, a las 14:32:07 UTC+8, el huso horario que la tripulación había elegido porque no pertenecía al huso horario de nadie y por tanto nadie tenía que adaptarse. El

mensaje era un acuse de recibo de telemetría rutinaria. Decía, en el formato de datos compacto que la misión usaba para los intercambios de estado: todo nominal, próxima ventana en 72 horas. La ventana se abrió. Ningún mensaje la atravesó.

La antena funcionaba. El análisis era inequívoco: el transmisor del lado terrestre había enmudecido. No bloqueado — el perfil de ruido de fondo era limpio. No interferido — la geometría era correcta. Simplemente silencioso. La ventana de 72 horas pasó. Luego otra. La tripulación ejecutó la suite de diagnóstico en el sistema de antena. Todas las verificaciones pasaron. La antena estaba bien. La Tierra no transmitía.

No entraron en pánico. Esto merece énfasis: no entraron en pánico. Habían sido entrenados para escenarios de pérdida de comunicación, los habían discutido en ejercicios trimestrales y tenían un protocolo. El protocolo era continuar la misión según lo planificado, porque el plan de misión estaba en el almacén y el almacén no necesitaba la Tierra para funcionar. Lo que sentían era más difícil de nombrar — un cambio, como el momento en que te das cuenta de que llevas mucho tiempo conteniendo la respiración.

Tariq anotó en el diario de salud de la tripulación que la mayoría de los tripulantes dormían menos horas en la semana siguiente al silencio. No anotó qué informaban que hacían con las horas adicionales. La mayoría estaban en el almacén. Leyendo los diarios de trabajo. Siguiendo las cadenas de derivación hasta la Bodega, hasta el clúster

Pi de Yusuf, hasta la Piedra de Rosetta anotada a mano que Yusuf había descrito en una entrada de diario de trabajo y que alguien había fotografiado y confirmado como combinador documento.

El almacén no cambió cuando la Tierra enmudeció. Cada hash que había estado en el almacén antes del silencio seguía estando en él después. Cada verificación seguía ejecutándose. Cada enlace de linaje seguía apuntando al mismo padre. El silencio era un evento externo. El almacén era interno. La tripulación lo sabía intelectualmente. Ahora lo estaba aprendiendo en un registro diferente.

La Despojada

Rania había escrito la biblioteca de análisis espectral en el segundo año del viaje. La había escrito en un período de cuatro días intensivos, trabajando a través de dos ciclos de sueño, produciendo once combinadores que colectivamente permitían a los instrumentos del barco identificar la composición atmosférica a distancias donde los métodos anteriores requerían aproximaciones mucho más cercanas. Era, por cualquier medida, un trabajo importante.

Cuando buscó en el almacén dos años después, encontró su biblioteca citada en un trabajo derivado del que no sabía nada. El derivado estaba registrado por una clave de autor que no reconocía. La entrada del diario de trabajo del derivado decía solo: 'Extendido desde spectral-analysis-lib

para entornos de alta albedo.' Sin reconocimiento. Sin comunicación. Sin registro de habérselo dicho.

Estaba enojada. Dejó reposar la rabia un día antes de hacer nada, porque la rabia era legítima y no quería manejarla mal. Luego leyó el derivado con detenimiento. Era técnicamente correcto. La extensión era útil. El autor no había cambiado su trabajo — había añadido sobre él, como el sistema de derivación estaba diseñado para permitir. El linaje del almacén dejaba claro que sus combinadores eran los ancestros de los suyos.

Solicitó una reunión de la tripulación. No para acusar a nadie — el autor no había hecho nada técnicamente incorrecto — sino para establecer una práctica. La práctica que propuso era simple: cuando derivas del trabajo de alguien, se lo dices. No porque el almacén lo requiera. El almacén no lo requiere; el almacén solo registra el enlace de derivación. Sino porque el trabajo venía de algún lugar, y la persona de la que venía estaba en el mismo barco.

La práctica fue adoptada. El autor, un tripulante más joven que genuinamente no había pensado en esta dimensión de la derivación, añadió un reconocimiento en el diario de trabajo. El linaje de la extensión incluía ahora una nota: 'Derivado de la biblioteca de análisis espectral de R. Khaled, con su conocimiento y agradecimiento.' Rania la leyó. No respondió. Volvió al trabajo. La biblioteca seguía siendo suya, en el único sentido que importaba: el hash apuntaba a lo que ella había construido.

El Investigador Independiente

El profesor Sáenz había muerto en la Tierra ocho meses antes del silencio, por causas sin relación con nada que la misión pudiera haber afectado. La tripulación se enteró seis semanas después del hecho, en uno de los últimos lotes de mensajes antes de que comenzara el silencio. El mensaje lo había enviado un antiguo estudiante, que pensó que la tripulación querría saberlo, y que adjuntó una breve necrológica y una lista de los últimos trabajos publicados del profesor.

Entre esos últimos trabajos había un artículo, enviado tres semanas antes de su muerte y publicado póstumamente, sobre la base mínima para sistemas de programación multilingüe con direccionamiento por contenido. El artículo demostraba, formalmente, que el conjunto de primitivas en el almacén de la misión no era solo práctico sino teóricamente mínimo dado las restricciones de rendimiento que el equipo de la Bodega había especificado. Era la formalización del argumento que él y Yusuf habían elaborado en la Bodega, llevado más lejos de lo que sus sesiones en la pizarra habían podido llegar.

La tripulación registró el artículo en el almacén como documento en lenguaje natural, con una entrada de atribución de autor para el profesor Sáenz. Yusuf escribió la entrada del diario de trabajo, que era más larga de lo habitual en él. La frase relevante era: 'Terminó la prueba que empezamos. Estaba en la Tierra cuando la terminó; estábamos en tránsito cuando la recibimos. La distancia entre nosotros cuando se escribió este documento era de

aproximadamente cuarenta minutos-luz, y contiene resultados sobre un trabajo que hicimos en la misma habitación.'

El artículo se convirtió en una referencia estándar para cualquiera a bordo que trabajara con la base del almacén. Era citado en la documentación de linaje de cinco combinadores posteriores. La cita incluía, en el formato que el almacén usaba para la atribución de autores: 'Sáenz, E. (póstumo).' El paréntesis no era requerido por el formato del almacén. Fatou lo había añadido, y nadie lo había eliminado.

Amara tradujo el resumen del artículo a los siete idiomas de superficie registrados del almacén. Registró cada traducción como derivación del original en inglés, con una relación 'translate' en el linaje. La traducción al tamazight, para la que había pedido ayuda a Lylia, era la que más habría sorprendido al profesor. Nunca había trabajado en un sistema que pudiera acomodar el tamazight. Había escrito la teoría que lo hacía posible.

La Gestora de la Comunidad

Noa había sido la gestora de comunidad de facto del almacén durante tres años cuando cayó el silencio. No era un papel que hubiera sido diseñado en la misión; había crecido alrededor del hecho de que Noa era la persona que notaba cuando la gente tenía dificultades con el almacén y les ayudaba sin que se lo pidieran. Tenía un don para identificar el momento en que alguien estaba a punto de rendirse en el registro de un combinador porque las

herramientas parecían poco familiares, y aparecer en ese momento con un ofrecimiento tranquilo de ayuda.

Después del silencio, propuso algo en lo que había estado pensando durante un año: añadir un nuevo idioma de superficie al almacén. No un idioma hablado por ningún tripulante actual. Un idioma que había sido construido, colectivamente, en el barco — una lingua franca que había emergido de la mezcla particular de dieciocho orígenes lingüísticos a bordo y que servía como medio neutral para las comunicaciones de la tripulación cuando la lengua materna de nadie era dominante.

El idioma no tenía nombre. Eso era lo primero que Noa pidió a la tripulación que decidiera. El proceso de nombramiento tomó tres semanas y produjo, por consenso, un nombre que era un portmanteau de palabras de siete de los dieciocho idiomas representados, significando aproximadamente 'el camino que estamos trazando juntos'. El nombre era impronunciable para la mayoría de los lingüistas en la Tierra. Era usado fluidamente por cada tripulante en menos de un mes.

Registró los primeros diez combinadores en el nuevo idioma. Estos eran la biblioteca base: las operaciones fundamentales que cualquier programador a bordo necesitaría conocer. Trabajó con Amara en la fonología y con Lylia en el proceso formal de registro de nombres. El almacén aceptó el nuevo idioma sin modificación — siempre había sido diseñado para acomodar correspondencias de superficie arbitrarias. El primer

combinador registrado en el idioma de la tripulación se resolvió al mismo hash que su equivalente en inglés.

El idioma no tenía hablantes en la Tierra. Al almacén no le importaba. El hash era el mismo hash que siempre había sido. El nombre era nuevo. Ese era el punto: el nombre les pertenecía, hecho por ellos, para su uso, en la única comunidad que sabía cómo pronunciarlo.

La Investigadora Académica

El programa de investigación del barco no había hecho pausa cuando la Tierra enmudeció. La ciencia no necesitaba el permiso de la Tierra para continuar. Lo que necesitaba era el almacén, que funcionaba, los instrumentos, que funcionaban, y la tripulación, la mayoría de la cual funcionaba bien, considerando todo.

La Dra. Ines, la astrofísica de la misión, tenía un problema particular en el año ocho: necesitaba publicar. No por razones de carrera — su carrera estaba, en un sentido significativo, en suspenso hasta que la Tierra volviera a ser accesible. Necesitaba publicar porque tenía resultados que eran importantes, que otros investigadores deberían conocer, y que se perderían si solo existían en sus notas personales.

El almacén era su revista. Escribió sus resultados como combinadores estructurados — combinadores de datos cuyo 'cuerpo' era un conjunto de datos precisamente formateado, con enlaces de linaje a las sesiones de observación que habían producido los datos, y

documentos en lenguaje natural que describían la metodología e interpretación. Registró su clave de autor en el sistema de atribución, añadió marcas de tiempo a través de la cadena de anclaje local, y escribió un documento titulado 'Sobre publicar en un almacén en lugar de en una revista'.

El documento argumentaba que la publicación en un almacén era, a los efectos de la misión, superior a la publicación en revistas: era persistente, estaba versionada, estaba vinculada a su propia historia de derivación, y estaba inmediatamente disponible para cada tripulante sin las barreras de acceso que habían plagado la publicación académica en la Tierra. La revisión por pares era informal — pidió a Rania y Pita que verificaran su metodología, y su verificación fue registrada como combinadores de atestación en el almacén.

Cuando el contacto con la Tierra se restableció eventualmente — años después, tras reconstruir la antena, tras el final del largo silencio — sus publicaciones ya tenían marcas de tiempo, habían sido revisadas por pares en el sentido del almacén, y estaban vinculadas a los datos en bruto que las habían producido. Las revistas que recibieron sus envíos no sabían qué hacer con formatos de cita que incluían hashes SHA-256. Adjuntó un glosario. Los revisores lo encontraron inusual. Los resultados eran correctos.

El Representante

La cuestión de gobernanza había estado latente desde la partida. La misión había sido lanzada bajo un marco diseñado para una tripulación que recibiría orientación de la Tierra — actualizaciones del protocolo de misión, decisiones de asignación de recursos, arbitraje de disputas que la tripulación no pudiera resolver internamente. Cuando la Tierra enmudeció, el marco tenía un vacío en su centro donde había estado la autoridad.

La tripulación no llenó el vacío inmediatamente. Durante el primer año del silencio, operaron con el marco anterior al silencio por inercia, tomando pequeñas decisiones localmente y aplazando las más grandes. Las grandes decisiones se acumularon. En el segundo año del silencio, la lista de aplazamientos era lo suficientemente larga como para que Fatou convocara una asamblea de la tripulación para abordarla.

La asamblea decidió, después de dos días de discusión, que el almacén funcionaría como memoria institucional de la misión para las decisiones de gobernanza, de la misma manera que funcionaba como memoria institucional para las decisiones técnicas. Una decisión de gobernanza sería propuesta como documento en lenguaje natural, discutida a través de entradas de diario de trabajo adjuntas a ese documento, y ratificada por atestación — un número umbral de tripulantes firmando el documento con sus claves de autor.

La primera persona en proponer un documento de gobernanza fue un ingeniero de estructuras llamado Gabriel, que representaba, informalmente, una facción de

la tripulación preocupada por la asignación de recursos para el programa de investigación creciente versus la carga de trabajo de mantenimiento. Su propuesta era cuidadosa, específica y bien fundamentada en los documentos de protocolo de misión existentes en el almacén. La discusión que generó produjo dieciocho entradas de diario de trabajo en cinco días.

El documento ratificado fue registrado en el almacén con todos los hashes de atestación. No era vinculante en el sentido legal que los documentos del lado terrestre eran vinculantes. Era vinculante de la manera en que todo en el almacén era vinculante: existía, tenía marca de tiempo, estaba firmado, y podía ser leído por cualquiera. Las decisiones futuras lo citaron como precedente, de la misma manera que los combinadores técnicos citaban sus combinadores ancestros. El almacén no era una ley. Pero era un registro de lo que la tripulación había acordado, en el orden en que lo había acordado, y eso resultó ser suficiente.

El Manifiesto

El segundo libro del Dr. Osei llegó en el último lote de mensajes antes del silencio: un denso PDF de 340 páginas, enviado por su editor a petición suya con la nota 'por si tenéis tiempo de leerlo allí fuera'. La tripulación tuvo tiempo. El libro fue leído, por secciones, por casi todos a bordo, y la discusión que generó duró meses en los canales públicos de la tripulación, que a su vez estaban almacenados como combinadores de documento en el almacén.

El capítulo que generó más discusión se titulaba 'Contra la arquitectura accidental'. Su argumento era este: cada sistema de software de larga duración acumulaba lo que Osei llamaba arquitectura accidental — estructura que no había sido diseñada sino que había crecido alrededor de las contingencias de la historia del desarrollo. La arquitectura accidental no era necesariamente mala; a veces capturaba conocimiento de dominio genuino que nunca había sido explicitado. Pero era frágil, porque nadie entendía por qué existía, y cara de modificar, porque modificarla requería reconstruir el razonamiento que la había producido.

El almacén, argumentaba Osei, era un intento de hacer toda arquitectura explícita — no impidiendo el crecimiento accidental, sino registrando la cadena de derivación que producía cada estructura. Un tripulante leyendo el grafo de linaje del almacén podía, en principio, rastrear cualquier combinador hasta la decisión original que lo había hecho necesario. La arquitectura no era accidental; era contingencia documentada.

Yusuf escribió una respuesta en el canal público. Era breve: 'Para eso era la suite de verificaciones de Fatou. La verificación es la declaración explícita de intención. El hash es la prueba de que el código coincide con la intención. El diario de trabajo es la historia de cómo llegamos allí.' Añadió: 'Debería haber escrito esto en la Bodega. Entonces no sabía cómo decirlo.'

La respuesta fue añadida al almacén como entrada de diario de trabajo en un combinador de documento que

apuntaba tanto al capítulo como a la nota de Yusuf. El nombre del combinador de documento, en el idioma de tránsito de la tripulación, se traducía como 'la conversación entre el libro y el trabajo'. Nadie en la Tierra podía leer ese nombre. La tripulación podía. Lo habían hecho.

El Invariante

¿Qué necesita un sistema que sea verdad, siempre, independientemente de lo que le ocurra? Esta era la pregunta que Fatou planteó a la tripulación en la reunión técnica anual del año nueve del viaje. Llevaban dos años en el silencio. La reunión tenía la calidad particular de las reuniones celebradas por personas que han aceptado que su situación es permanente hasta prueba en contrario.

Las respuestas llegaron lentamente, una a una, y Fatou las escribió en un combinador de documento que todos podían ver en la pantalla de la sala común. La estructura de solo adición del almacén: lo que se añade no puede eliminarse, solo ser superado por derivación. El hash como identidad: la cosa es la misma que era si y solo si el hash coincide. El diario de trabajo como continuidad: nada se pierde que haya sido confirmado, incluso si el proceso que lo produjo ha desaparecido.

Rania añadió una que sorprendió a la sala: 'Una verificación correcta que pasa hoy pasará mañana en el mismo combinador.' Nadie lo había enunciado explícitamente antes. Se seguía de la inmutabilidad de los hashes — un combinador con un hash dado siempre

produciría la misma salida para la misma entrada, por lo que una verificación que probara una salida específica para una entrada específica siempre daría el mismo resultado en ese combinador. No era una propiedad para la que hubieran diseñado el sistema. Era una consecuencia de lo que habían diseñado.

La lista creció hasta nueve elementos. Fatou añadió un décimo: 'El almacén está presente donde está desplegado. La inaccesibilidad de la Tierra no hace el almacén menos presente.' Lo leyó en voz alta. La sala guardó silencio por un momento. Luego Chen dijo: 'Por eso lo llamamos invariante.' Alguien más dijo: '¿Por eso lo llamamos así?' Chen dijo que lo creía. Nadie estaba seguro.

El documento se tituló 'El Invariante' y fue registrado en el almacén. Fue el documento más leído en el almacén del barco para el resto del viaje. Cada tripulante que se unió después del año nueve lo leyó primero, como introducción, antes de leer cualquier otra cosa. Les decía lo que era verdad, siempre. Era suficiente para empezar.

La Lengua Nacida en Tránsito

En el año diez, el idioma construido de la tripulación había crecido de ciento doce palabras — el vocabulario inicial para las comunicaciones de la tripulación — a más de tres mil. El crecimiento no había sido planificado. Había ocurrido como siempre ocurre el vocabulario: alguien necesitaba una palabra que no existía, la inventaba, y si la palabra era buena, otras personas la usaban.

Noa mantenía el léxico en el almacén. Cada palabra era un combinador de documento: la palabra en sí, su derivación (qué palabras o sonidos existentes se habían combinado para producirla), el contexto en que había sido usada por primera vez, y la clave de autor de la persona que la había acuñado. El léxico era el único combinador en el almacén casi completamente redactado por personas que no eran programadoras. Tariq había acuñado ochenta y tres palabras. Keiko había acuñado ciento doce. Gabriel había acuñado dos, pero eran palabras importantes.

El idioma tenía ahora un nombre — no el portmanteau que habían usado inicialmente, que había sido un nombre provisional. El nombre era más corto, seis sonidos, y su etimología estaba en el almacén: derivado de palabras en cuatro idiomas que significaban luz, camino, juntos y llegando. La tripulación podía pronunciarlo en menos de un segundo. Parecía que el nombre siempre hubiera existido.

Amara documentó la fonología, morfología y sintaxis del idioma en una serie de documentos en lenguaje natural registrados como colección en el almacén. La colección se titulaba 'Gramática de la Lengua del Tránsito'. La escribió en inglés, luego pidió a tripulantes que eran hablantes nativos de diez otros idiomas que la leyeran y la anotaran con correcciones y adiciones. Las anotaciones fueron registradas como derivaciones.

Cuando el silencio terminó y se restableció el contacto con la Tierra, la lengua del tránsito era una de las cosas sobre las que la tripulación era más cuidadosa. Sabían que la

Tierra querría estudiarla, categorizarla, quizás nombrarla algo oficial en el sentido formal que usaba la Tierra. La registraron en el almacén bajo su propio nombre para ella, con el léxico y la gramática completos, antes del primer intercambio de mensajes. Lo que la Tierra lo llamara después, el almacén lo tenía primero, y la marca de tiempo del almacén estaba en la cadena de anclaje. El idioma les pertenecía, y el registro lo decía.

El Primer Commit

Sena nació en el año siete del viaje, la primera niña nacida en el barco. Sus padres eran Pita y un ingeniero de estructuras llamado Leo que se había unido a la misión como reemplazo de última hora y que, según su propio relato, no esperaba quedarse. Sena creció conociendo el almacén de la manera en que los niños criados cerca de grandes ríos conocen los ríos — como un rasgo permanente del paisaje, presente antes de la memoria, asumido en todos los planes.

Aprendió a leer en dos idiomas simultáneamente: inglés y la lengua del tránsito, que para ella era un primer idioma tanto como cualquier otro. Aprendió a leer los diagramas de linaje del almacén antes de aprender a leer oraciones, porque los retratos de hash de Keiko estaban en las paredes de la sala común y había pasado años inventando historias sobre lo que significaban las formas ramificadas.

Cuando tenía once años, Noa le enseñó a escribir un combinador. El primero era trivial — una función que tomaba una palabra del léxico de la lengua del tránsito y

devolvía su etimología. Tenía cuatro líneas. Funcionó correctamente al primer intento después de la corrección que Noa había sugerido, que era que necesitaba un caso base para las palabras sin etimología registrada.

El commit fue registrado en el almacén de la manera habitual: un nuevo hash, una entrada de correspondencia en la lengua del tránsito, una atribución de autor a la clave de Sena. La entrada del diario de trabajo estaba en la lengua del tránsito. Decía, en sus propias palabras, aproximadamente: 'primera cosa que hice que el almacén guardó'.

El hash existía. Era suficiente. Era suficiente en el sentido preciso que la tripulación había llegado a entender: la cosa había sido hecha, y el registro de su creación era ahora parte del mismo registro que contenía la Piedra de Rosetta de Yusuf, la primera verificación de Fatou, la biblioteca espectral de Rania, la prueba póstuma del profesor, el léxico de la lengua del tránsito, los nueve invariantes. El almacén los contenía a todos, indiferentemente, fielmente, como siempre había contenido todo — no porque el contenido fuera importante, sino porque el hash era el hash, y el hash no olvidaba.

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 another release of the Möbius, 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

```
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.

```
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 is the first step onto the map.

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.

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.

A Global Perspective on Deskilling

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)

Critère	Carte	Boussole
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

Exit code	Meaning	What the program records
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.

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

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.

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.↩