# HyperDoc - a hypermedia substrate for knowledge workers

Konrad Hinsen
konrad.hinsen@cnrs.fr

Centre de Biophysique Moléculaire (UPR4301 CNRS)
Rue Charles Sadron, 45071 Orléans Cédex 2, France
and
Synchrotron SOLEIL, Division Expériences
B.P. 48, 91192 Gif sur Yvette, France

**Abstract**

HyperDoc is a research project about growing a hypermedia substrate for knowledge workers that integrates code as a representation of knowledge. This paper explains the motivation for this work, discusses the research and development strategy, and provides a concise description of its current and possible future state.

## 1 Goals, values, development strategy

The **audience** for HyperDoc is **knowledge workers**, i.e. professionals who consult and contribute to networks of collective knowledge. This includes scientists, engineers, medical professionals, journalists, and many others. HyperDoc must therefore be able to support complex distributed networks of diverse media, as well as both manual and automated processing. On the other hand, it can also tolerate a steeper learning curve than a substrate for knowledge management at the personal or small-group scale.

Designing for a clearly identified audience is important if one wants to preserve the audience's agency over the evolution of the substrate. As a warning example, the World-Wide Web (WWW) was initially developed for knowledge workers as well [Berners-Lee 1990]. However, it was quickly adopted to other domains of application, whose stronger economic backing lead to their dominance in the evolution of WWW standards. These standards and the software that implements them (browsers) have become so complex that only a handful of large corporations retain agency over the WWW. The needs and interests of knowledge workers are no longer influential.

## 1.1 Code as knowledge

The major **innovation** attempted in HyperDoc is **integrating code** into hypermedia systems, which have traditionally been limited to passive media, with rare exceptions such as Apple's HyperCard.[1] Code in HyperDoc has the double role of a **representation of knowledge**, meaning source code written for communicating computational methods to human readers, and a **medium for making tools**. The latter are mostly small and situated, in the spirit of shell scripts but supporting interactivity and visual information representations beyond text. Tools can also be of pedagogical nature, as in explorable explanations [Victor 2011]. The two roles of code are complementary: a tool can be valuable both via its utility and via the knowledge expressed in its source code.

The motivation for a closer integration of code with other hypermedia is the complementarity of different hypermedia types in transmitting knowledge between humans. Algorithms and formal relations between entities are better expressed in code than in plain language. Today, the lack of good integration of code with prose leads to the coexistence of two representations of the behavior of a software system: code for the machine, and an informal, incomplete, and often inaccurate description in plain language for humans. Moreover, the correspondence between the code and its informal summary is almost impossible to verify once code grows beyond a few hundred lines.

The focus on code as a knowledge representation also places a high value on **code transparency and understandability**, favoring clear and simple situated code over general or generalizable and reusable code, in opposition to software engineering principles such as "Don't Repeat Yourself" (DRY). Situated code also means that prior work serves mainly as an inspiration or as a starting point for customization, rather then as a re-usable library that becomes a dependency. In a system geared towards code as a knowledge representation, re-use should be limited to well-understood and well-tested libraries that are stable and that readers can be expected to be familiar with. In other words, libraries from the infrastructure pace layer [Brand 2018]. This can be seen in analogy to science communication: a journal article typically presumes that readers have textbook knowledge (i.e. infrastructure) in the field, but explicitly cites and summarizes other journal articles.

## 1.2 The ecosystem of collective knowledge

In the spirit of networks of collective knowledge, which grow rather than being designed, HyperDoc adopts a strategy of **continually evolving the**

[1] Source code on its own is often presented as hypertext in code editors and IDEs, where e.g. a click on a function name moves the reader to the definition of that function. However, these developer tools do not permit the embedding of source code into a diverse hypermedia network.

**state of the art**, rather than attempting a top-down design or a clean-slate approach. To a practicing knowledge worker, adopting HyperDoc should feel like integrating a new software library into their workflow, rather than like switching to a new programming language or a new development toolbox. This aspirational goal is clearly not yet achieved by the current prototype, which supports a single programming language.

Collective knowledge evolves on multiple pace layers [Brand 2018]. Knowledge grows via the integration of new information. This happens at the fastest pace layer, in which individuals and teams work on a time scale of a few years, publishing scientific articles, technical reports, and similar very focused documents. Knowledge is consolidated by a continuous community effort of verifying and comparing contributions from the fastest layer. This consolidation layer advances on a time scale of about a decade, and is expressed in publications such as review articles and monographs. A third layer refines the consolidated knowledge further into consensual knowledge that is recorded in textbooks for teaching, and shared with decision makers in politics and industry. It operates on a time scale of a few decades. Even slower layers operate on time scales of centuries to millenia, evolving the world views shared by entire cultures.

Computing technology is only a few decades old and still undergoes rapid change. Code as a knowledge representation is feasible at the fastest pace layer, but remains a challenge for knowledge consolidation. The most long-lived software technologies we have today are those that have written standards and multiple actively maintained implementations. This unfortunately excludes popular programming languages such as Python.

HyperDoc aims at **supporting the pace layers of knowledge consolidation**, and therefore the digital-era equivalents of scientific publications, review articles, and university textbooks. The use of Common Lisp as the main implementation language for HyperDoc is motivated by practical considerations discussed in section 2.2, but also intended as a reminder about the importance of long-term stability in programming systems for contributions to collective knowledge.

Being a good citizen of existing and emerging knowledge ecosystems also implies **building bridges** to other technologies, and helping them build bridges to HyperDoc. This is the role of the hyperbook layer of the substrate (see section 2.1.2). The two bridges included in the current prototype integrate Wikipedia, a repository for consensual knowledge, and Federated Wiki, a knowledge repository in the agile pace layer of team collaboration. Bridges to both slower and faster pace layers allow the exploration of knowledge migration between layers.

These two bridges consider their targets as abstract knowledge repositories

rather than as Web sites. They support a unified user interface (UI), shared with other hyperbooks, that offers different affordances than the repositories' native UIs. One important affordance is the cross-repository link graph. Wikipedia pages in particular tend to become hubs in this graph as authors refer to them for background reading. Back links from Wikipedia pages thus serve as a topic index across the whole substrate.

Another affordance provided by HyperDoc is the unified multi-pane presentation of material in Miller columns, allowing the reader to put related material side-by-side, as opposed to the Web's dominant single-page views where clicking on a link replaces the page by another one. Miller columns have been used for many years in Federated Wiki and Glamorous Toolkit. Parallel display of multiple hypermedia items is particularly important for code, which often needs to be viewed along with its documentation, or along with the results that it produces. Moreover, the smaller views encourage a hypermedia structure consisting of many small and focused pages, which in turn allows readers to follow personalized paths through the hypermedia graph according to their interests and prior knowledge, giving them increased agency [Liu and Almeda 2025].

A final affordance is a computational interface to knowledge repositories, allowing readers to analyze their content by writing code in playgrounds (see section 2.1.1).

At first sight, alternative views on pre-existing knowledge repositories can seem like a form of platform enclosure. But HyperDoc is not a platform with modernist aspirations of universality. HyperDoc provides one view out of many on Wikipedia and Federated Wiki. It does not attempt to displace their native UIs. On the contrary, every HyperDoc view on a Wikipedia or Federated Wiki page has a button for accessing the native UIs. Inversely, HyperDoc's publication units are also available as Git repositories for use with other tools or substrates.

## 2 The current state: a research prototype

The current implementation of HyperDoc is a research prototype that focuses on two aspects of a substrate for knowledge workers:

1. Integration of code into hypermedia graphs
2. Building bridges to hypermedia knowledge repositories

Another practically important aspect is deliberately excluded: building bridges to existing programming systems. I will outline ideas for dealing with this aspect in section 3. The main reason for excluding it initially is the enormous implementation effort, which exceeds the capacities of an unfunded single-person research project.

An on-line demo is available for exploration. At the time of this writing (April 2026), it is frequently slow to respond due to a heavy load, probably from AI crawlers. For better reactivity, install the demo server on your own computer.

## 2.1 Substrate architecture

The substrate consists of three layers. The object graph layer provides a general navigation and interaction infrastructure for information items. The hyperbook layer defines the general properties of hypermedia publishing units, and manages links between them. This is where bridges to existing knowledge repositories are implemented. The HyperDoc layer defines a HyperDoc as a specific kind of hyperbook, consisting of text, code, and tool pages plus named data items.

### 2.1.1 Object graph

Even though the intended application domain of the HyperDoc substrate is hypermedia, the fact that it integrates executable code implies that it must also handle the inputs and outputs of code, and that means arbitrary data items. The base layer of HyperDoc is therefore a user interface to a general object graph, i.e. a graph of in-memory data items that can refer to each other. This is the data model introduced by Lisp in the 1950s, on which most dynamic programming systems are based today [Sitaker 2016].

The user interface that HyperDoc provides for the object graph is an object inspector in the Lisp and Smalltalk tradition. HyperDoc's object inspector is strongly inspired by the Smalltalk object inspector in Glamorous Toolkit, sharing in particular the use Miller columns and the plugin architecture for implementing class-specific views. Hyperbooks and HyperDocs, described below, are implemented as classes to which custom views are attached to create the user interface. HyperDoc authors can use the same mechanisms for adding views and user interfaces for domain-specific data types.

Active exploration of computational knowledge repositories often requires writing and executing code snippets, e.g. for extracting specific data, comparing information from different sources, etc. The object graph inspector therefore proposes a *playground*, inspired by a common Smalltalk tool of the same name. A playground is a text editor whose contents are persisted across sessions, on the user's computer. It is used to write code snippets, optionally with comments, that are useful to the user in a particular context. The HyperDoc object inspector manages one playground page per class. Its initial contents can be defined in code, via a method implementation for the class. Note that the code execution in playgrounds is disabled in the on-line demo for safety reasons, as the current implementation has no sandboxing

features.

### 2.1.2 Hyperbooks

A hyperbook is a set of named hypermedia items called pages. The word "page" suggests text, but a page can represent any kind of hypermedia. Every hyperbook has a unique ID, which is expected to be a Universal Resource Identifier (URI) (a convention that is not yet respected in the current prototype). Each page has an ID as well, which must be unique inside the hyperbook. Both hyperbooks and pages also have titles for presentation to human readers. Links connect pages, the target of a link is defined by a hyperbook ID and a page ID.

The hyperbook layer of the substrate contains support code for implementing hyperbook types, such as HTML rendering code and link management. A basic hyperbook implementation is included as well, both for reference and for use in the hyperbook layer's documentation. Such a basic hyperbook consists of HTML pages whose titles are also their IDs.

In addition to HyperDoc, explained in section 2.1.3, there are currently five additional hyperbook implementations:

- An interface to Wikipedia
- An interface to Federated Wiki
- Three interfaces to code units in a Common Lisp image:
  - classes
  - functions
  - systems (units of development and distribution, what most languages today call *packages*)

The code-related hyperbooks are an illustration of the generality of the hyperbook abstraction, as well as a concrete mechanism to integrate code into a hypermedia graph.

### 2.1.3 HyperDocs

A HyperDoc is a hyperbook that supports three types of pages. Text pages (example) are written in HTML or Markdown, both being extended with HTML custom elements. Code pages (example) are standard source code files whose first line is expected to be a comment that becomes the page title. Tool pages (example) are defined in code; the user interface is implemented in terms of HTML elements via a dataflow infrastructure. Playground pages (example), based on the playgrounds implemented at the object inspector level, allow HyperDoc authors to propose code snippets to the reader for modification and execution, e.g. as exercises in a tutorial. A HyperDoc can also contain named data items that are usable and discoverable from code in

other HyperDocs.

Code pages have special rendering support in the HyperDoc substrate. Function, class, and variable symbols are clickable, and refer to the corresponding definitions. Moreover, code annotations can be added as inert functions (they are stripped away by the compiler) whose arguments are evaluated by the renderer to derive links. This makes it possible to add hyperlinks anywhere in the code.

Code examples (examples) are zero-argument functions. The renderer adds a button that permits one-click execution, with the object inspector showing the result.

Text pages can transclude functions, classes, and inspector views on arbitrary objects. There is also a HTML custom element for inserting the result of a computation into text.

## 2.2 Implementation substrates

The main implementation substrate for the current HyperDoc prototype is **Common Lisp**. Its two relevant properties are its good **support for interactivity and introspection**, and its **long-term stability**. Support for interactivity and introspection is immediately relevant for HyperDoc because most the code-specific features (section 2.1.3) depend on it. Implementing HyperDoc in a less supportive language would require significantly more effort, comparable to implementing an IDE. Long-term stability matters for the lower pace layers of collective knowledge (see section 1.2).

A HyperDoc unit is realized as an ASDF System whose files are kept in a version-controlled repository. It is thus a standard Common Lisp software package, with additional conventions about the file layout. Since HyperDoc units must interact with the substrate, the latter imposes an additional dependency: the ASDF system "hyperdoc". This is kept as small as possible, and uses only a small number of well-known stable dependencies of its own.

In the current prototype, every hypermedia item is represented by an in-memory object, which is often a proxy to some other representation, such as a file or a network resource. The Web-based object inspector is implemented using CLOG, a Common Lisp framework for Web applications that provides a bridge between Lisp and in-browser JavaScript, which is a secondary implementation substrate for HyperDoc. It is mainly used for rendering information in the browser using the numerious JavaScript libraries available for such purposes.

### 2.3 Priorities

The initial focus is on consulting rather than authoring; the current prototype is read-and-run-only. There are many existing tools for authoring hypermedia and code (though they tend to be distinct), and the first generation of HyperDoc adopters are expected to continue to use whatever tools they are already familiar with. It is consulting and exploring code-as-knowledge that is badly supported today, both in terms of tooling and in terms of practices, including in particular authoring practices that favor comprehensibility. Recent work on explorable explanations [Brusilovsky 1994; Victor 2011] and explainable software systems [Nierstrasz and Girba 2022] is the main source of inspiration.

In the long run, integration of individual and collaborative authoring tools with tools for consultation and exploration is highly desirable. A good hypermedia substrate should support collaboration at the three relevant time scales: (1) real-time collaboration, as supported by tools such as Etherpad or CodiMD, (2) asynchronous team collaboration, as supported by a wide range of tools, e.g. version control systems, e-mail, or wikis, (3) long-term community-scale collaboration, mediated by information substrates such as books, journals, and Web sites.

## 3 Outlook: HyperDoc as a protocol

While Common Lisp is a convenient implementation platform for exploring design choices with minimal technical overhead, it is clear that in the long run, HyperDoc must support a large variety of programming languages if it wants to become a useful substrate for knowledge workers who are familiar with these languages and depend on code written in them. In principle, HyperDoc functionalities can be implemented for any programming language, in much the same way as Integrated Development Environments (IDEs) can be built for any language. The implementation effort, however, varies significantly between languages.

The main challenge for a future polyglot HyperDoc is not an implementation for any single language, but the co-existence of code written in different languages. A researcher whose preferred language is Python 3 might well want to write a small tool that compares two priorly published models, one written in Python 2 and the other in the R language. One aspect of this problem is the lack of tools and techniques for managing the integration domain [Kell 2009]. Another aspect is the time scale mismatch between the evolution of many programming languages and the evolution of collective knowledge, as explained in section 1.2. Today, scientific knowledge that has been expressed exclusively in Python 2 code is at risk of being lost because this language version is no longer maintained since 2020.

The long-term vision for HyperDoc is an **implementation as a protocol**, each unit being realized as a more encapsulated artifact such as a Unix process or a Web server. This would allow combining different implementation technologies. The bridging protocol between such units is a major challenge, and would best be supported by a lower-level substrate such as proposed in [Kell 2025]. Minimal bridging could be achieved by data exchange via standard serialization formats such as JSON. However, many use cases also require some form of remote procedure call and the possibility to hold references in one runtime environment to mutable data structures that live in a different runtime environment. An added complication is the occurence of very large datasets and long-running computations in knowledge repositories. This adds performance requirements that network-based protocols cannot satisfy. Finally, the HyperDoc protocol must remain reasonably simple to implement, as otherwise the many implementations required by the diversity of programming languages will never happen. The HyperDoc protocol should be implementable in "dead" but still relevant languages such as Python 2, and in domain-specific languages such as Fortran or APL (though possibly via foreign-function interfaces).

The most promising path towards such a HyperDoc protocol is again growing it by composing existing protocols. HTTP(S) plus WebSockets are sufficient for communication between HyperDoc units (implemented as Web servers) and the user interface of the object inspector. Interaction between HyperDoc units are the most difficult part. There are many existing protocols that could be useful (serialization via JSON or XML, content-addressable data stores such as IPFS, messaging protocols such as XMPP, publishing protocols such as ATProto, etc.), but no single one has widespread support, and no single one is sufficient to cover all requirements.

## 4 Acknowledgments

I would like to express my gratitude to the five reviewers of the Substrates 2026 workshop for their thoughtful, in-depth, and constructive feedback. I hope to have taken it into account adequately in this revised version.

## References

Berners-Lee, T. 1990. *Information Management: A Proposal*.

Brand, S. 2018. Pace Layering: How Complex Systems Learn and Keep Learning. *Journal of Design and Science*.

Brusilovsky, P. 1994. Explanatory visualization in an educational programming environment: Connecting examples with general knowledge. *Human-Computer Interaction*, Springer, 202–212.

Kell, S. 2009. The mythical matched modules: Overcoming the tyranny of

inflexible software construction. *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications - OOPSLA '09*, ACM Press, 881.

Kell, S. 2025. Substratus unicus. *Substrates 2025*.

Liu, S. and Almeda, S.G. 2025. Agency Among Agents: Designing with Hypertextual Friction in the Algorithmic Web. https://arxiv.org/abs/2507.23585.

Nierstrasz, O. and Girba, T. 2022. Making Systems Explainable. *2022 Working Conference on Software Visualization (VISSOFT)*, IEEE, 1–4.

Sitaker, K.J. 2016. The memory models that underlie programming languages. http://canonical.org/~kragen/memory-models/.

Victor, B. 2011. Explorable Explanations. https://worrydream.com/ExplorableExplanations/.