Coding computational laws: 20 recommendations for public administrations

Raphaël Monat[a] Liane Huttner[b]

Abstract

Public administrations are steadily digitalizing all their procedures. In particular, computational laws – such as taxes and benefits – are increasingly implemented within computers, enabling scalable, automated computations. These computer implementations have four key specificities: they are *critical software* at the *intersection between law and computer science*, that will be *updated regularly* by legal changes, and have a *long lifespan*, counted in decades. Thus, great care should be taken to avoid any issue in these specific legal implementations. Building upon years of studying and coding computational laws, both in administrations and as new research products, we propose 20 recommendations to ease the development and maintenance of legal implementations. These recommendations aim at being understandable for lawyers and computer scientists alike.

1 Introduction

In the last three decades, all states have steadily digitalized their public services, due to the constant availability of computers, as well as the efficiency and cost cuts digitalization can provide. The European Union, through its ambitious "Digital Decade Policy Programme" (European Parliament and European Council, 2022) aims in particular at reaching 100% availability of "key public services" of all member states by 2030.

However, this digitalization brings challenges in terms of accessibility and transparency of public services. In this work, we consider the implementation of computational laws, that is, the computer code running laws precisely describing computations – for example to describe taxes or benefits. We argue that these computer programs have four key specifities. First, they are to be considered **critical software**, as they can have dramatic societal impacts, e.g. by unfairly denying benefits to families in need. While critical software usually concerns embedded systems within avionics or power plants with life-threatening consequences, we argue that the studied systems can have life-changing impacts too. Second, they are **regularly updated** through series of patches coming from legislative processes. Third, these implementations have a really **long lifespan**. Some have been developed in the late 90s and are supposed to run for decades to come, meaning they need to be well designed and maintained to last so long. Last but not least, translating computational laws into computer code is a complex process, that **requires close collaboration between computer scientists and lawyers**. These past six years, we have been able to work on legacy systems used in production by administrations, and to participate in an interdisciplinary, international research project aiming at designing next-generation tools easing the implementation and maintenance of public code. Our journey started through a work aiming a reverse-engineering parts of the legacy compiler used for the French income tax. We have been able to establish a close collaboration with the administration in charge of producing and running the income tax computations. As a side benefit, we were able to observe the current processes established by this administration. Thanks to this collaboration, we

[a]Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[b]Centre d'Etudes et de Recherche en Droit de l'Immatériel, Univ. Paris-Saclay, F-92330 Sceaux, France

have been able to successfully complete our reverse engineering and create a modern compiler (Merigoux, Monat, & Protzenko, 2021) to replace their legacy tool. This compiler is now being integrated back into the public administration. Following this work, we have taken part in designing Catala (Huttner & Merigoux, 2022; Merigoux, Chataing, & Protzenko, 2021), a next-generation computer language tailored to implement computational laws in a transparent, lawyer-friendly fashion. Preliminary experiments made by the team on real-world legal texts over the past few years have validated the key design decisions behind Catala, and opened several interesting questions around legal interpretations made by developers (Merigoux et al., 2023, 2024; Monat et al., 2024).

Building on this line of research we have been pursuing, our discussions and observations both with academics and administrative civil servants, we share in this essay what we believe to be the most important takeaways when coding computational laws. This essay can be considered both as a set of recommendations and as a basis for further discussion. It is primarily targeted at people who are writing new implementations of the law, but can also be used as a checklist for people working on already existing codes and for scholarly discussions on the topic.

Limitations. There are many different kind of laws, diverse ways to code the law and different types of actors. Our essay targets the implementation of computational laws, as done by public administrations to provide public services to citizens. This article does not target systems aiming at helping judges, implementations for private interests, laws that are not computational per se (such as nationality acts or contract law). Our experience is limited to Western societies.

Outline. A summary of our twenty recommendations is provided in Figure 1. These recommendations are split in six different kinds, on which we elaborate in the following sections, explaining why these recommendations are essential and how one can implement them. We finish by briefly discussing related work.

2 Be aware of the specificities of implementing computational laws

> **Recommendation 1:** Coding the law is an interdisciplinary work between legal experts and computer scientists that can only be reached through mutual trust and respect.

Coding the law is a difficult task that require close collaboration between lawyers and computer scientists. This collaboration can only be successful if both parties respect and trust the other. Both parties need to look at the other discipline with an open-mind and accept difficulty. This can prove challenging since law and computer science function very differently.

On the one hand, programming leaves no room for ambiguity. A computer cannot understand a programmer directly. The programmer gives instructions to a computer through programs, written in specified programming languages. The computer will only obey code instructions to the letter. Ambiguities cannot be resolved by computers, they have to be taken care of during the programming part by a human. A wide variety of programming languages exist, each with their sweet spot, for example in terms of ease of coding and long term maintenance.

On the other hand, laws are not a simple set of rules needing implementation by computer scientists. Complexity, ambiguities and unplanned situations form an entire part of the legal system – which that has been functioning for millenniums. Unclear, imprecise and contradictory laws are perfectly normal, because of their social, political and philosophical functions. In this respect, legal scholars and lawyers have developed complex tools and concepts. They have discussed for decades – and even centuries – how to interpret the law, how to combine laws that contradict each other, and to what extent applying the law requires active choices. Coding

Figure 1: Summary of recommendations

Be aware of the specificities of implementing computational laws

1. Coding the law is an interdisciplinary work between legal experts and computer scientists that can only be reached through mutual trust and respect.
2. Since legal codes are critical software, use tools and techniques to ensure high-assurance of your code.
3. Ensure that the codebase has a structure similar to the law which will allow you to maintain it.

Make your implementation abide by the law

4. Your implementation needs to abide by local data protection and administrative laws.
5. Guarantee transparency of your code.
6. Make sure you have the complete set of laws you are implementing.
7. Report, tag and document any legal interpretation or additional hypothesis you are using in your implementation

Use the right programming tools

8. Build a program that is deterministic and well-defined, without undefined behaviours such as division by zero, out-of-bound array accesses.
9. Choose the right datatypes carefully:
    - Use arbitrary-precision numbers to represent monetary units and avoid rounding imprecision.
    - Beware of date-related computations. They need to handle leap years, and durations expressed in months or years are not precise.

Ease day-to-day technical development

10. Test your code rigorously with lawyers, and address changes in the law.
11. Version your code.
12. Put together continuous integration that ensures that your codebase is identified, backed up and tested regularly. Ensure that production releases are bundled and published automatically.
13. Separate developing concerns to facilitate maintenance
14. Setup a centralized approach to track known issues in your codebase

Ensure your project will last

15. Have a clear documentation for incoming users and developers.
16. Assess bus factor: number of developers with critical knowledge not shared with the rest of the team.
17. Beware of proprietary solutions, both at the hardware and the software level.

Facilitate public interaction

18. Aim for explainable decisions that can be understood by citizens.
19. Allow individuals to appeal results of automated computations.
20. Use a bug tracking platform which also allows individuals to notify the problems they are confronted with.

the law is impossible without knowing about these characteristics of the law and having a basic understanding of the already existing answers.

It is only through this mutual understanding that coding the law can be done. Lawyers and computer scientists will thus have to interact, in order to detect and resolve ambiguities arising from the law during its translation into code.

Recommendation 2: Since legal codes are critical software, use tools and techniques to ensure high-assurance of your code.

Critical software are software whose bugs can have dramatic consequences on human lives. These typically are software used in planes, trains, hospitals or nuclear power plants. These software pieces are traditionally validated by independent bodies, checking compliance with standards. For example in civil avionics, the EASA & FAA check compliance with DO-178C (RTCA, 2011).

We argue that software implementing laws can be classified as critical software as well. For example, social benefits and tax collection depend almost entirely on the calculation done by the computers and receive little human supervision. Any issue in the implementation can have tremendous impact on citizens or even state funding. We provide some example of notorious failures in Examples 1 and 2, and refer concerned readers to Redden et al. (2022) for a survey of systems that were so faulty they have been canceled.

Thus, having a high assurance that the code being run corresponds to the law is paramount. Treating programs that implement the law as critical software means that considerable resources must be invested to ensure the implementation works as intended. As in transport, energy or health, there can be a sub-classification of the criticality of legal software. Creating such a classification is left as future work.

Example 1: Phoenix automated payroll failures for Canadian civil servants (Mockler, 2018)

The Phoenix automated payroll system was developed by IBM, and deployed in 2011, with the goal of modernizing the pay of Canadian civil servants. The program should have provided "$70 million in annual savings by centralizing pay operations". However, the replacement of the legacy system failed, and more than half of all Canadian civil servants suffered pay issues from the Phoenix automated payroll system, resulting in years of fixing incurred financial issues for civil servants. What should have been an economical measure transformed into "$2.2 billion in unplanned expenditures" for the Canadian government.

Example 2: Dutch childcare benefits scandal

Through a failing fraud detection system used between 2005 and 2019, the Dutch Tax and Customs Administrations wrongly accused more than tens of thousand of parents to making fraudulent benefits claims. These accusations lead to reimbursement demands, having tremendous financial consequences on beneficiaries. The fraud detection system has been described by the Dutch Data Protection Authority as "unlawful, discriminatory and improper" (Dutch Data Protection Authority, 2020). The scandal culminated with the resignation of the government in 2021.

Bouwmeester (2023) searches to understand reasons behind the failure of this system, and highlight unused control mechanisms of the legislative and judiciary branches.

Recommendation 3: Ensure that the codebase has a structure similar to the law which will allow you to maintain it.

Legal texts have a specific structure, usually consisting commonly consisting of a base case refined by exceptions spread out throughout its description. This structure, whose core can be encoded as default logic (Brewka & Eiter, 2000; Lawsky, 2017), is not straightforward to encode using modern programming languages. Programmers can thus be tempted to introduce discrepancies between code and law, or at least to structure the codebase differently from the law. However, these implementations are then hard to maintain in sync with legislative texts in the long term, as programmers may have trouble pinpointing the code impacted by a legal modification. We have observed real-life cases where these effects turn out to be detrimental as they accumulate over the years: most computational laws tend to evolve at least yearly, accumulating large implementation overheads due to inadequate initial implementation choices.

3 Make your implementation abide by the law

Recommendation 4: Your implementation needs to abide by local digital and administrative laws.

In addition to the laws the authors are implementing, authors should be keenly aware that some "meta-laws" regulate machine implementations of the law. States and legislators have adopted rules that will apply to the activity of coding the law, and not taking those laws into account might end up in creating an illegal implementation. This distinction is essential and the computation must respect both of these categories. At a high-level, there are two categories of meta-laws applying to implementations: administrative law, and the rapidly expanding field of digital law, encompassing in particular data protection, cybersecurity and artificial intelligence. These laws are defined at different levels: some of them are national, and others stem from international treaties. As these laws vary substantially depending on the country, we highlight the general concepts relating to these laws and the current trends.

Administrative law. Computations made and used by administrations (such as tax computations and social benefits) are also concerned by administrative laws. In particular, we highlight what we believe is a major step regarding transparency of implementations: since 2017, French administrations are required to publish the source code of their implementation (Legifrance, 2016). The state now maintains a list of implementations developed by administrations (Direction Interministérielle du Numérique, 2023). If some are missing, citizens can seize an administrative body (called CADA, 1978) to require implementations to be made public. We believe transparency is paramount to maintain public trust and ensure a high level of software quality. These administrations also have to inform citizens whenever an algorithm is used to take a decision about them. Citizens can also request more specific information about the way algorithmic computations have impacted their situation. Most importantly, the Constitutional Council has expressly forbidden the use of self-learning program for public algorithms (Conseil Constitutionnel, 2018).

Digital law. Data protection law deals with the processing of personal data through automated or non-automated means. If the computer program uses data that is linked to the identity of an individual, then it must respect these laws. This will be the case most of the times, since the laws will be applied to a specific person. Furthermore, personal data is often defined very broadly, in all existing legal instruments. In the GDPR for example, "personal data" means any information relating to an identified or identifiable natural person, who can be identified, directly or indirectly (article 4 (1)). An IP address is, for example, considered as

personal data. The application of data protection law means that the programmer must respect a certain number of principles. Most importantly, the concept of necessity in data protection law may restrain the legality of the implementation. In Europe, for example, and as the CJEU stated in several cases (notably C-524/06, Heinz Huber ; C-582/14, Breyer and C-73/ 16, Peter Puškár), necessity means that the processing is "suitable for achieving the objectives pursued by them and whether there is no other less restrictive means in order to achieve those objectives". This is a strict conception of necessity. It means that the computation of the law must be the best way to achieve its implementation.

Cybersecurity and artificial intelligence laws have been recently introduced, e.g by the European Union with its Cyber Resilience and Artificial Intelligence Acts announced in 2024. We believe some software used by administrations are subject to these laws, although it is a bit early to tell: both acts will apply in 2026-2027 at the earliest.

Recommendation 5: Guarantee transparency of your code.

Note that even if transparency of the computational code may not be required by your local administrative laws, we strongly recommend to make your code public. This will improve transparency of administrative processes and tackle criticism about opaque behaviors. It will also incentive to keep good quality code.

More broadly, at an ethical level, we believe that when implementing the law, programmers must have a sense of the criticality of the code they implement and the potential impact (both positive and negative) they can have on citizens.

Recommendation 6: Make sure you have the complete set of laws you are implementing.

Going back from the meta-laws to the laws targeted by an implementation, we also want to stress that finding the complete set of laws specifying the implementation is a paramount first step before implementing anything.

Recommendation 7: Report, tag and document any legal interpretation or additional hypothesis you are using in your implementation.

Imprecision, contextuality, need of interpretation, contradiction are inherent to the concept of law itself. These characteristics of the law are major challenges to overcome for their implementation into computer code. Indeed, computers can only execute a fixed, specific sequence of instructions, where all ambiguities have been resolved.

Computer implementation of the law will thus require choices. This is similar to any other implementation of the law, be it by judges or the administration. We recommend all choices to be clearly documented in the source code and tagged as such. This will allow for easy search, but it also serves more important purposes: accountability and transparency over these choices.

Sometimes, the choices may appear as self-evident to domain experts or programmers. Sergot et al. (1986) describe for example choices that have been made in the interpretation of the British nationality act, stating that it is "usually possible to identify the intended interpretation with little difficulty". Even if it is the case, we believe that the mere existence of the interpretation needs to be documented and clearly established for the sake of transparency.

We provide below examples of ambiguities to be resolved.

Example 3: Ambiguity of sale in Section 121

In Section 121 of the US Tax Code, the word "sale" is used in many different ways. It can refer to the sale of the property that will provoke the exclusion of the gain of Section 121, or it can refer to other sales, happening earlier or later in team. Encoding all these situations in a single "sale" variable in the computation would be insufficient, because the

computer will not be able to discriminate between them. As a consequence, we need to define different variables that are seemingly all expressed by the term "sale" in the law. An implementation (Gesbert et al., 2024) needs in particular to differentiate sales that have been made by individuals before they were filing jointly (under section 121 (B)), needing to refer previous sales through variable `other_section_121a_sale`.

Example 4: An example of micro-choice (Merigoux et al., 2023)

Merigoux et al. (2023) argue that already choosing which inputs citizens will be asked to fill amounts to a form of legal interpretation made by developers, which they call "micro-choices". For the sake of illustration, they consider a part of the French housing benefits, where one eligibility criterion is "the first day of the calendar month following the third month of pregnancy in respect of a child of rank three or more and the last day of the month preceding that in which the child reaches his or her second birthday" (Legifrance, 2019). The authors argue that some choices may not respect privacy, and could even ask for medical data (like the "presumed date of the beginning of the pregnancy" used by French health professionals). In the end, they conclude that a good input is to ask users whether their situation consists in being before or after the first day of the third month of pregnancy, ensuring a good level of privacy which eases compliance with locally applicable privacy laws.

4 Use the right programming tools

Recommendation 8: Build a program that is deterministic and well-defined, without undefined behaviours such as division by zero, out-of-bound array accesses.

We recommend using the right programming tools, to ensure that your programs are well-behaved, and avoid unexpected behaviors that may happen randomly, either in time on the same machine, or across different machines.

Our first recommendation is to ensure that your codebase does not contain undefined behaviors. By definition, the programs containing those behaviors may do anything, and the result may also depend on which computer the program is run. Usual undefined behaviors include division by zero and out-of-bounds array accesses. Some unsafe languages allow silent failures with easy recovering operations. While it may seem appealing in the short term, we highly recommended against it. This is a recipe for introducing some weird operations in your codebase, which may stay until no one knows why it should happen anymore. We observed this kind of issue in real-world cases used in production.

Recommendation 9: Choose the right datatypes carefully:

- Use arbitrary-precision numbers to represent monetary units and avoid rounding imprecision.
- Beware of date-related computations. They need to handle leap years, and durations expressed in months or years are not precise.

Our second recommendation concerns the computer representation of decimal numbers. As computer memory is finite, not all numbers can be explicitly represented. A popular representation is floating-point arithmetic, which offers a good performance-precision ratio for tasks such as numerical computing. Floating-point operators are well studied by experts, who have been able to prove that the relative rounding error of each operation can be upper-bounded. We refer the interested readers to "Handbook of Floating-Point Arithmetic" from Muller et al. (2018), for an in-depth coverage of this topic. This scientific community has also developed

tools that can estimate rounding errors of a given program, if floating-point computations are the option you finally choose. However, we warn the readers that a lot of expertise is needed to avoid cascading rounding errors which may lead to catastrophic results; an example is provided in Example 5. Basic numbers such as 0.1 are not exactly representable in binary floating-point arithmetic.

Example 5: An example of catastrophic imprecision in floating-point arithmetic (Muller et al., 2018)

In their "Handbook of Floating-Point Arithmetic", Muller et al. (2018) provide an example of a sequence of computations (provided below) which converges towards 6 using real mathematical numbers. However, computations of this sequence using traditional floating-point arithmetic offered by computers will converge to a value of 100, resulting in a catastrophic imprecision.

$$\begin{cases} u_0 = 2 \\ u_1 = -4 \\ u_n = 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}} \end{cases}$$

If there are no specific legal guidance on the precision of decimal computations, nor floating-point experts to help you, we strongly recommend using arbitrary-precision numbers to represent decimal numbers you may use (monetary units, ...) in order to avoid catastrophic results. An example of arbitrary-precision numbers library is GNU's MP (GMP Granlund and GMP Development Team, 2015). If you really need to rely on floating-point numbers, you may want to carefully choose between using binary and decimal representations (Cowlishaw, 2003).

To a lesser extent, we would like to warn our readers about date-related computations. First, the case of leap years should not be forgotten. More importantly, durations expressed in months or years will have a length depending on the starting date, and computing these durations may even be ambiguous, as shown in Example 6. As we argued before, we seek to proscribe any ambiguous computation, and advise readers to seek legal guidance to clarify these cases. We refer the interested reader to the work of Monat et al. (2024) formally defining date arithmetic and providing tools to detect ambiguous computations.

Example 6: Ambiguous computations in date arithmetic (Monat et al., 2024)

Let us consider the date defined as one month after March 31st. Since April has only 30 days, you can either consider the result to be April 30th or May 1st, depending on the "rounding mode" you choose. Some bodies of law choose another approach, specifying that a month is a duration defined as 30 days.

A similar case happens when you want to compute the age of someone who is a "leaper", i.e., born a February 29th – depending on the country, leapers will come of age on March 1st or on February 28th.

5 Ease day-to-day technical development

In addition to their criticality, another defining aspect of legal implementations is their really long lifespan: unless the implemented body of law is radically changed, the implementation may have started 20 years ago and can live – and will have to be kept updated – for decades to come. This maintenance will be a significant challenge, and any step where time has been invested to ease maintenance will have tremendous positive impact in the years to come. This section offers advice to reduce those costs, some of them being standard software engineering techniques. We refer the reader to Thomas and Hunt (2019) for an in-depth reference about

software engineering and project development.

Recommendation 10: Test your code rigorously with lawyers, and address changes in the law.

We recommend establishing a test suite for your implementation. These tests describe what your implementation is supposed to return for specific input cases. These cases can be either real or fictitious cases. Running those tests will provide an easy way to check that you have not made breaking changes into your codebase. It will also make sure during your design phase that you all agree on some examples. These tests should be manually established by legal experts to make sure they abide by the law. They should cover most, if not all of the possible cases encountered by your implementation.

Every time the law is updated, those tests will have to be thoroughly reviewed to address any changes. Adding tests for new cases is also recommended. We cannot emphasize this strongly enough: the correct update of testcases is critical to ensure smooth maintenance and to detect potential bugs as early as possible.

Recommendation 11: Version your code, and ensure that production releases are bundled automatically.

Using a version control system such as "git" will bring many benefits to your team and your development workflow. In particular, it provides means to work as a team on the same codebase: you can make modifications in your own space, and then share them with the rest of the team so they can review those before you add them to your official codebase. It keeps a history of your software, which always comes in handy when you search for the source of a regression. Finally, it allows to back up your codebase in a server.

Recommendation 12: Put together continuous integration, ensuring that your codebase is identified, backed up and tested regularly. Ensure that production releases are bundled and published automatically.

We suggest you add continuous integration to your project. This mechanism will regularly fetch your codebase, and perform some sanity checks on it, for example by verifying it can be run and that the test suite mentioned above returns the expected results. This process is automatic and periodic – it can for example be configured to be run each time you make a new change or each night. It will allow for early detection of breaking changes in your codebase, provided your test suite is covering the changes.

An added benefit of continuous integration and code versioning is that releases and distribution of the different versions of your software can be done automatically. Automatic releases will not take human development time, and avoid human errors in packaging.

Recommendation 13: Separate developing concerns to facilitate maintenance

We recommend separating concerns in your codebase: the legal part of your code should not be written in the same repository as, for example, its interface. In practice, the different interfaces (for example, those used by other service in your administration, or by citizens) can even be handled by another team. This separation of concerns also means you have to carefully consider which components will be exposed to other administrative services. Let us assume your legal codebase contains a number of different functions. Some of these functions are for your internal use only: you may decide to change them a lot across different versions, remove them, replace it with others... If you make public every function of your codebase, those internal functions could be used by other services. This would restrict your maintenance freedom as you would not be able to change those internal functions, at risk of breaking the code of other parts

of your administration. Thus, we recommend to only make the necessary functions exposed when you distribute your component.

Recommendation 14: Setup a centralized approach to track known issues in your codebase.

A bug tracking platform is another tool that should help software maintenance. It provides a way to centralize the issues you have discovered about your project, categorize and discuss those issues as well as assign whoever should fix it. It also provides a way for everyone to see how the project is going on. If you do not centralize issues or some of their data, for example by discussing some issues over email, or if you keep the list of issues as a local file, it will have dire consequences at the inevitable time when your team will change. In addition, having a central platform may trigger inputs or discussions from your teammates and help resolve the issue. We recommend that known corner cases of your implementation be also added to your bug tracker.

6 Ensure your project will last

Recommendation 15: Have a clear documentation for incoming users and developers.

All teams have a life, where some members may leave and others may come. Accidents may also happen, creating abrupt and unexpected departures. We recommend to have all processes clearly documented in written documents. Writing this documentation will take time, but it may help pinpoint some issues in the current workflow, and will help with onboarding newcomers!

Recommendation 16: Assess bus factor: number of developers with critical knowledge not shared with the rest of the team.

Alongside this process, we recommend assessing the "bus factor" of your team: the number of members with critical pieces of knowledge that is not shared with any other member of the team and whose knowledge would disappear in case of an accident. This assessment should happen regularly, and depends on the the dynamics of your team evolution, and the potential risks of its members. Whenever possible, we advise to document the identified critical pieces of knowledge to reduce risk.

Recommendation 17: Beware of proprietary solutions, both at the hardware and the software level.

We are strongly wary of proprietary solutions, where a company provides access specific hardware or software they develop and license to you. These solutions may be appealing in the short term, by providing external expertise and ready to run, potentially specialized solutions. However, you will lose expertise and you may end up locked in a solution developed by a single company. While your administration will have to continue its work and implementation, proprietary companies have their own priorities, can go bankrupt or significantly increase their fees in the long term.

7 Facilitate public interaction

Recommendation 18: Aim for explainable decisions that can be understood by citizens.

In order to guarantee a high level of trust and accountability, we recommend that computational implementations should be explainable: citizens should be able to trace and follow how

the computation reached a given amount or decision, and how this has changed compared to previous computations.

In France, administrations are required by the law to provide explanations[1], although explainability is still a research topic and we have yet to see concrete and compelling examples of explainable decisions in production.

Recommendation 19: Allow individuals to appeal results of automated computations.

When an explanation is not satisfying, individuals should be able to appeal the result of the computational implementation. This is for example codified by the European Union's GDPR Recital 71 which mentions that people can "obtain an explanation of the decision reached after such assessment and to challenge the decision" European Parliament and European Council, 2016.

Recommendation 20: Use a bug tracking platform which also allows individuals to notify the problems they are confronted with.

As a last resort, we recommend providing an open, bug tracking platform tracking appeals and potential misbehaviors identified by citizens. This will improve transparency. It may also help in fixing bugs not for a single household but ensure the fix is backpropagated to all affected households in similar situations.

8 Related work

Analyzing failures and detecting errors. Redden et al. (2022) study 61 cases of canceled automated public services across Europe, UK, North America, Australia and New Zealand. From this study, they emit ten recommendations in order to avoid further cancellations, but more importantly to avoid deployment of tools having negative impact on citizens. Escher and Banovic (2020) define method to detect errors in a benefit screening tool of the Pennsylvanian state, finding several case where the screening tool would advise not to apply to benefits citizens would have been eligible to. Tizpaz-Niari et al. (2023) propose a debugging method for the US tax preparation software, which is applied to strengthen open-source tools by finding several accountability bugs and missing eligibility conditions. Goutagny et al. (2025) present a method to automatically find all interpretation conflicts within a Catala program, and apply it on the implementation of the French housing benefits.

Other guides and recommendations. van Eck et al. (2022) provide a way to audit legal implementations through a three-way perspective: legal, computer science, and through the lens of accountability obligations. Some audits have already been done in the Netherlands, and the guidelines have been adopted by the state. Andrews (2022) describe guidelines to design trustful and transparent AI systems for the case of the public sector. Chignard and Guerry (2019), as senior members of the DINUM French administration (in charge of the digital policy for the state), provide a guide for all French administrations, explaining what kind of algorithms may bed used in the public sector, ethics and responsability issues as well as local laws regarding general algorithmic transparency. This guide builds upon a position paper (Chignard & Penicaud, 2019) from high-ranking French civil servants identifying specifics challenges of public sector algorithms, providing recommendations to design accountable automated procedures. The Organization for Economic Co-opereration and Development (OECD) published two reports in 2020 related to public code. The first report (Organisation for Economic Co-operation

[1] "For these decisions, the supervision officer ensures that [...] able to explain, in detail and in an intelligible form, to the concerned person the way in which the processing has been implemented with regard to them." (Legifrance, 2018)

and Development, 2020) is a whitepaper envisioning future tax systems (dubbed "Tax Administration 3.0"), where tax systems are closely integrated with "taxable events", improving the overall efficiency of administrations. The second report Mohun and Roberts (2020), alongside Diver (2021), advocate for "rules as code", meaning that legislative processes should already produce code instead of laws that can only be interpreted by lawyers, and then implemented as code. Similarly, a New Zealand Government report advocates to co-design legislation and the corresponding machine-consume version. While we agree that better cooperation between lawyers and computer scientists are required to improve public implementation of computational law, we have several reservations this approach. First, the interaction with implementations of laws predating this paradigm is not considered, and it seems unlikely that all laws targeting e.g. taxes would be canceled to start a clean slate for the purposes of a new development approach. Second, it is our understanding that legislators, in some cases, barely have the time to draft laws: assuming they have the skills to write computer code, it is unlikely that they would have time to do it during processes. Third, this approach reminds us of tech companies developing non-critical software which can afford to "move fast and break things". However, we agree with Mohun and Roberts (2020) on the need for close interdisciplinary collaboration and their following declaration "An approach emphasising the use of a multidisciplinary team and the co-creation of human and machine-consumable rules appears most likely to deliver". Through an experimental study, Guitton et al. (2025) find that providing frameworks to improve implementations of automatically processable regulations lead participants to creating better designs, suggesting that organizations should be mandated to use such guides in order to minimize potential issues within their systems.

On proprietary solutions. Bouras et al. (2013) provide guidelines helping public administrations choosing between open-source and proprietary software. Note that the Organisation for Economic Co-operation and Development (2020) warn of the potential risks of delegating work to the private sector, and proprietary solutions, in particular with "the risk of commercial lock-in, where one company holds proprietary access to the rules and can unfairly leverage a service or platform". The Commons Strategies Group (2012), in "The Wealth of the Commons" go further and advocate that public administration should use free software, in particular to keep citizen's trust and avoid delegating critical infrastructure to foreign companies.

Legacy & Modernization. Bellotti (2021) describes how to maintain legacy computer systems, and provides modernization strategies and insights. Starting from the European Commission "no legacy principle" – where systems older than 15 years would have to be replaced – Irani et al. (2023) study the impact of legacy systems and the digital transformation of European public administrations. They highlight in particular that "legacy systems are frequently associated to vendor lock-in situations" (in case of proprietary software), which upon modernization creates compatibility issues. Bozeman (2003) explores the causes of failure of the 1990-1996 modernization of IRS (US) tax system and describe cultural changes performed in the IRS following this failure.

Coding the law. There is a long line of research studying, both from theoretical and practical perspectives, how laws can be implemented, and what impact it can have.

In the field of law, Pierre Catala pioneered the proposition that programmers establish a set of "invariants" on which the structure of the computer program will rest. These "invariants" are the concepts that will probably never change because they form the basis of the particular legal statute (Catala, 1998). Understanding the statute through these invariants can be a good start, though it is not sufficient in practice.

Our recommendation 3 corresponds to the early work of Bench-Capon and Coenen (1992)

recommending to establish isomorphism, i.e., a well-defined correspondence between an implementation and the corresponding legal knowledge-based system.

Hoffmann-Riem (2020) warn against the risks of translating legal rules – which can be interpreted in various ways, to allow leeway through human factors – into computer code where no such ambiguity can be kept; this can thus lead to radical changes in how laws are applied. They base their discussion on German legal provisions regulating automated administrative decisions.

Ranchordas and Scarcella (2021) highlight the inequalities that digitalization of public services can bring, as full digitalization reduces accessibility to parts of the population unable to use, or access, a working computer. The authors argue that these inequalities can be further strengthened when private companies provide intermediate software used to interact between citizens and state (such as tax preparation software in the US).

Escher et al. (2024) study the behavior of fifteen teams of students (being either in computer science or law curriculum), tasked with implementing parts of the US Bankruptcy code in the JavaScript programming language. Only two teams succeeded in creating a faithful implementation of the law. What is more, most computer science students were confident in their tool and a significant part were willing to replace human judges by their tool. Through this study, the authors thus advocate for greater care to be taken during production of legal software, as the disciplinary separation between computer science and law is a huge source of errors and mistranslations. This work experimentally confirms our Recommendation 1 about the need for deep cooperation between legal experts and computer scientists involved in implementing computational laws.

The Catala programming language (Huttner & Merigoux, 2022; Merigoux, Chataing, & Protzenko, 2021), named after Pierre Catala, ensures a structural correspondence between law articles and their implementation through literate programming. Literate programming means that the code is always linked to the law it is based on, which improves transparency of the implementation and simplifies maintenance.

Besides the scalability benefits stemming from the computer implementation of computational laws, new research shows it can also benefit citizens and administrations alike. Merigoux et al. (2024) are exploring new interfaces to make the legal computations more explainable to non-expert citizens; Goutagny et al. (2025) present a method to automatically find all interpretation conflicts.

9 Conclusion

In this paper, we advocate for 20 recommendations that will improve the state of public legal code. We believe our most important takeaway is that implementing the law is inherently interdisciplinary work between legal experts and computer scientists. It requires a high expertise in both domains, that can only be reached through a deep cooperation based on mutual trust and respect.

Acknowledgments. We are grateful to James Barnes, Pierre Goutagny, James Grimmelmann, Sarah Lawsky and Denis Merigoux for their helpful comments on early versions of these recommendations. We thank the whole Catala team for the many discussions which contributed towards this paper and the valuable feedback we received.

Bibliography

Andrews, P. (2022, October). *Designing for legitimacy*. https://apolitical.co/solution-articles/en/designing-for-legitimacy

Bellotti, M. (2021). *Kill it with fire: Manage aging computer systems (and future proof modern ones)*. No Starch Press.

Bench-Capon, T. J., & Coenen, F. P. (1992). Isomorphism and legal knowledge based systems. *Artificial Intelligence and Law*, *1*(1), 65–86. https://doi.org/10.1007/BF00118479

Bouras, C., Kokkinos, V., & Tseliou, G. (2013). Methodology for public administrators for selecting between open source and proprietary software. *Telematics Informatics*, *30*(2), 100–110. https://doi.org/10.1016/J.TELE.2012.03.001

Bouwmeester, M. (2023). System failure in the digital welfare state: Exploring parliamentary and judicial control in the dutch childcare benefits scandal. *Recht der Werkelijkheid*, *44*(2), 13–37. https://doi.org/10.5553/RdW/138064242023044002003

Bozeman, B. (2003). Risk, reform and organizational culture: The case of IRS tax systems modernization. *International Public Management Journal*, *6*(2), 117–144. https://doi.org/10.1016/S0723-1318(04)13008-9

Brewka, G., & Eiter, T. (2000). Prioritizing default logic. In *Intellectics and computational logic: Papers in honor of wolfgang bibel* (pp. 27–45). Springer. https://doi.org/10.1007/978-94-015-9383-0_3

CADA. (1978). *Commission d'accès aux documents administratifs*. https://www.cada.fr/

Catala, P. (1998). *Le droit à l'épreuve du numérique: Jus ex machina*. PUF.

Chignard, S., & Guerry, B. (2019). *Guide des algorithmes publics*. https://etalab.github.io/algorithmes-publics/guide.html

Chignard, S., & Penicaud, S. (2019). *With great power comes great responsibility: Keeping public sector algorithms accountable*. https://github.com/etalab/algorithmes-publics/blob/master/20190611_WorkingPaper_PSAAccountability_Etalab.pdf

Conseil Constitutionnel. (2018, June). *Decision 2018-765 DC, ğ71*. https://www.conseil-constitutionnel.fr/decision/2018/2018765DC.htm#numero-considerant-71

Cowlishaw, M. F. (2003). Decimal floating-point: Algorism for computers. *16th IEEE Symposium on Computer Arithmetic (Arith-16 2003), 15-18 June 2003, Santiago de Compostela, Spain*, 104–111. https://doi.org/10.1109/ARITH.2003.1207666

Direction Interministérielle du Numérique. (2023). *Les codes sources du secteur public*. https://code.gouv.fr/sources/

Diver, L. (2021). *Interpreting the rule(s) of code: Performance, performativity, and production*. https://law.mit.edu/pub/interpretingtherulesofcode/release/4

Dutch Data Protection Authority. (2020). *Werkwijze belastingdienst in strijd met de wet en discriminerend*. https://autoriteitpersoonsgegevens.nl/actueel/werkwijze-belastingdienst-in-strijd-met-de-wet-en-discriminerend

Escher, N., & Banovic, N. (2020). Exposing error in poverty management technology: A method for auditing government benefits screening tools. *Proc. ACM Hum. Comput. Interact.*, *4*(CSCW), 064:1–064:20. https://doi.org/10.1145/3392874

Escher, N., Bilik, J., Banovic, N., & Green, B. (2024). Code-ifying the law: How disciplinary divides afflict the development of legal software. *Proc. ACM Hum. Comput. Interact.*, *8*(CSCW2), 1–37. https://doi.org/10.1145/3686937

European Parliament and European Council. (2016). *GDPR recital 71*. https://eur-lex.europa.eu/eli/reg/2016/679/oj/eng

European Parliament and European Council. (2022). *European union digital decade policy programme 2030*. https://eur-lex.europa.eu/eli/dec/2022/2481/oj

Gesbert, L., Lawsky, S., & Merigoux, D. (2024, January). *Catala implementation of section 121 of US tax code*. https://github.com/CatalaLang/catala-examples/blob/36055e91fdd936eac56d747cd40012dcd58f0403/us_tax_code/section_121.catala_en

Goutagny, P., Fromherz, A., & Monat, R. (2025). Cutecat: Concolic execution for computational law. *ESOP 2025*, *15695*, 31–61. https://doi.org/10.1007/978-3-031-91121-7_2

Granlund, T., & GMP Development Team. (2015). *Gnu mp 6.0 multiple precision arithmetic library*. https://gmplib.org/

Guitton, C., Druta, V., Landuyt, D. V., Bellemans, J., Tamò-Larrieux, A., & Mayer, S. (2025). A validation study of frameworks for responsible automatically processable regulation. *AI & SOCIETY*. https://doi.org/10.1007/s00146-025-02479-4

Hoffmann-Riem, W. (2020). Legal technology/computational law: Preconditions, opportunities and risks. *Journal of Cross-disciplinary Research in Computational Law*, *1*. https://journalcrcl.org/crcl/article/view/7

Huttner, L., & Merigoux, D. (2022). Catala: Moving Towards the Future of Legal Expert Systems. *Artificial Intelligence and Law*. https://doi.org/10.1007/s10506-022-09328-5

Irani, Z., Abril-Jimenez, R., Weerakkody, V., Omar, A., & Sivarajah, U. (2023). The impact of legacy systems on digital transformation in european public administration: Lesson learned from a multi case analysis. *Gov. Inf. Q.*, *40*(1), 101784. https://doi.org/10.1016/J.GIQ.2022.101784

Lawsky, S. B. (2017). A logic for statutes. *Fla. Tax Rev.*, *21*, 60. https://doi.org/10.2139/ssrn.3088206

Legifrance. (2016, October). *Article 2, loi nř 2016-1321 du 7 octobre 2016 pour une république numérique*. https://www.legifrance.gouv.fr/jorf/article_jo/JORFARTI000033202948

Legifrance. (2018). *Alinéa 2, loi nř 78-17 du 6 janvier 1978 relative à l'informatique, aux fichiers et aux libertés*. https://www.legifrance.gouv.fr/loda/article_lc/LEGIARTI000037823131

Legifrance. (2019, September). *Article d823-20 du code de la construction et de l'habitation*. https://www.legifrance.gouv.fr/codes/article_lc/LEGIARTI000038878895

Merigoux, D., Alauzen, M., Banuls, J., Gesbert, L., & Rolley, É. (2024, January). *De la transparence à l'explicabilité automatisée des algorithmes : comprendre les obstacles informatiques, juridiques et organisationnels* (tech. rep. No. RR-9535). INRIA Paris. https://inria.hal.science/hal-04391612

Merigoux, D., Alauzen, M., & Slimani, L. (2023). Rules, Computation and Politics. Scrutinizing Unnoticed Programming Choices in French Housing Benefits. *Journal of Cross-disciplinary Research in Computational Law*, *2*(1), 23. https://inria.hal.science/hal-03712130

Merigoux, D., Chataing, N., & Protzenko, J. (2021). Catala: A programming language for the law. *Proc. ACM Program. Lang.*, *5*(ICFP), 1–29. https://doi.org/10.1145/3473582

Merigoux, D., Monat, R., & Protzenko, J. (2021). A modern compiler for the French tax code. In A. Smith, D. Demange, & R. Gupta (Eds.), *CC'21* (pp. 71–82). ACM. https://doi.org/10.1145/3446804.3446850

Mockler, P. (2018). *The phoenix pay problem: Working toward a solution*. Senate Canada. https://publications.gc.ca/pub?id=9.859900&sl=0

Mohun, J., & Roberts, A. (2020). Cracking the code: Rulemaking for humans and machines. https://doi.org/10.1787/3afe6ba5-en

Monat, R., Fromherz, A., & Merigoux, D. (2024). Formalizing Date Arithmetic and Statically Detecting Ambiguities for the Law. *ESOP'24*, *14577*, 421–450. https://doi.org/10.1007/978-3-031-57267-8_16

Muller, J., Brunie, N., de Dinechin, F., Jeannerod, C., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N., & Torres, S. (2018). *Handbook of floating-point arithmetic (2nd ed.)* Springer. https://doi.org/10.1007/978-3-319-76526-6

Organisation for Economic Co-operation and Development. (2020). *Tax administration 3.0: The digital transformation of tax administration*. OECD. https://doi.org/10.1787/ca274cc5-en

Ranchordas, S., & Scarcella, L. (2021). Automated government for vulnerable citizens: Intermediating rights. *Wm. & Mary Bill Rts. J.*, *30*, 373. http://wm.billofrightsjournal.org/wp-content/uploads/2019/05/V30I2_07_RanchordasScarcella.pdf

Redden, J., Brand, J., Sander, I., Warne, H., Grant, A., & White, D. (2022). Automating public services: Learning from cancelled systems. *Cardiff, UK: Data Justice Lab, Cardiff University*. https://carnegieuk.org/publication/automating-public-services-learning-from-cancelled-systems/

RTCA. (2011). DO-178C, software considerations in airborne systems and equipment certification.

Sergot, M. J., Sadri, F., Kowalski, R. A., Kriwaczek, F., Hammond, P., & Cory, H. T. (1986). The british nationality act as a logic program. *Commun. ACM*, *29*(5), 370–386. https://doi.org/10.1145/5689.5920

The Commons Strategies Group. (2012). *The wealth of the commons. a world beyond market & state*. Levellers Press. https://wealthofthecommons.org/essay/public-administration-needs-free-software

Thomas, D., & Hunt, A. (2019). *The pragmatic programmer: Your journey to mastery*. Addison-Wesley Professional.

Tizpaz-Niari, S., Monjezi, V., Wagner, M., Darian, S., Reed, K., & Trivedi, A. (2023). Metamorphic testing and debugging of tax preparation software. *SEIS@ICSE'23*, 138–149. https://doi.org/10.1109/ICSE-SEIS58686.2023.00019

van Eck, M., Lokin, M., Klip, M., Bössenecker, G., Oldeman, C., van Doesburg, R., Klop, A., & Gort, S. (2022, July). *Legitimaat 1.0.3: A working method for conducting third-party research into the use of algorithms by a government organization*. https://minbzk.github.io/LegitiMaat/