

Test-Driven Development for Technology Policy

**APPLYING SOFTWARE ENGINEERING
PRINCIPLES TO POLICY DEVELOPMENT**

Neal Parikh
Ginny Fahs
Brandie Nonnecke



ASPEN TECH POLICY HUB

Executive Summary

The design of rockets and space shuttles has been one of the chief science and engineering achievements of the United States and all of humankind. An underappreciated aspect of these inspiring projects is the design and engineering of the software used to operate the rockets. Every time a shuttle fires up, a multi-billion dollar piece of equipment, the lives of astronauts, and the dreams of a nation all depend on the rocket's software working perfectly. This software cannot be tweaked after the fact if it does not work as intended — it has one shot, the first time it is used, to execute flawlessly.

It was in this demanding environment that the field of “software engineering” was born. The term, coined by the MIT mathematician Margaret Hamilton in the 1960s when working on the Apollo missions, refers to the subfield of computer science concerned with how best to create high-quality software.¹ “Quality” can refer to many things in the context of software, but at the very least, the software must do what it was actually intended to do and should not behave in unexpected ways.

While this standard may seem straightforward on the surface, it is difficult to achieve. Software even close to the quality used in space missions is the exception; even major tech companies see features fail multiple times per day. Decades of work in software engineering have produced tools that reliably improve the quality of the software that engineers produce. Many of these are process or project management innovations, not technical tools, and have proliferated in the past twenty years to become standard in the software industry.²

Like software, technology policy frequently does not work as intended: it does not fully address the issue it is concerned with or it has negative unintended consequences. Many people from the policy world acknowledge this problem; it was one of the issues that prompted the creation of the [Aspen Tech Policy Hub](#), where we conducted this work. We believe that some of the difficulties in crafting good technology policy can be mitigated by adapting what are now widely accepted ideas from the field of software engineering. Developing software was once as ad-hoc and error-prone as technology policy development often is now, but the policy process can and should evolve just as the software development process did.

The general idea of incorporating engineering thinking and methods into tech policy development and implementation is not new; it has been championed and successfully used in government as recently as the Obama administration.³ Jennifer Pahlka, founder of [Code for America](#) and a former Deputy U.S. Chief Technology Officer, wrote that tech's “user-centered, iterative, data-driven practices can work on policies themselves, and eventually on our laws.”⁴ Describing

her experiences working with technologists and the methods they use, Cecilia Muñoz, former head of the Domestic Policy Council in the Obama administration, called the use of software engineering principles in project management “the most transformative thing I’ve seen in my eight years in government.”⁵

Pahlka and Muñoz were primarily concerned with the effective implementation or delivery of an existing policy, while our work is concerned with improving the process of technology policymaking — and, in turn, tech policy itself. Our main goal is to help policymakers produce technology policy that is “robust,” meaning that it actually addresses the issue they are trying to address and that potential unintended consequences have been considered. We advocate using a particular engineering methodology called “test-driven development” for the development of technology policy, broadly defined to include corporate policy; federal, state, and local legislation; and other government functions.

Test-driven development does not require the use of any engineering tools or software code; rather, it is a process that any policymaker can follow.

This short guide explains software testing and test-driven development; shows how concepts from software testing can be adapted to the policymaking process; illustrates what this process might look like on sample technology legislation; and offers guidelines to help policymakers apply this method to their own policy projects.

What is Software Testing?

To write complex software, engineers break down a program into smaller components. For example, consider an engineer developing a website that includes a signup form that lets a new user register an account. One of the elements of this form may be a field asking for the user’s phone number.

For each element of a website, engineers write tests to assert that the code does what it is supposed to do. In the case of the phone number field, the tests would consist of entering a range of potential inputs that should or should not be accepted by the form. Coming up with these tests involves brainstorming possible phone numbers that a user might try to submit through the form and deciding whether they should or should not be accepted. Each test is implemented as a small piece of code.

For example, the engineer might want to test that the form accepts both “(650) 723-2300” and “650-723-2300” in the phone number field. The engineer may also decide that the form should reject “911”, a real US phone number but one that is not valid for a user account on this website; “+44-20-7925-0918”, be-



Photo by Tine Ivanič on Unsplash

cause the company does not allow international signups yet; and “ABCDEF,” because it is not a number. If the engineer were designing a different product, like the phone app on an iPhone, it would be important to ensure that the software allows 911 rather than rejects it.

What is Test-Driven Development?

Test-driven development (TDD) is an approach to software testing that (counterintuitively) advises writing tests first, then writing code for the actual program such that the code satisfies the requirements laid out in the test.⁶ To run the tests, engineers use testing software, which automatically plugs test data into the program and determines whether the program accepts or rejects it as expected. If all the tests pass, the engineer has more confidence that the code is written correctly.

Code is general and abstract; tests are specific examples. For example, the code to satisfy the tests above may accept any input matching the patterns “(DDD) DDD-DDDD” or “DDD-DDD-DDDD”, where D represents any digit. This would allow someone to sign up with the number “(212) 555-1234,” for instance.

Let’s pretend the engineer who is coding the form now wants to update the website to add support for users in the United Kingdom. He or she would start by changing the existing “+44-20-7925-0918” test, which originally said that

such a phone number should be rejected, to determine that the number *should* work in the program. The engineer would then add additional tests for UK numbers to codify what types of numbers the form should allow and disallow (for example, the engineer might still want the form to reject 999, the UK’s 911 equivalent). Then, the team would update the code for the signup form accordingly, and rerun all the tests for both US and UK numbers. This system would ensure that, in the process of adding support for users signing up with UK numbers, the updated code does not inadvertently break its previous support for signing up US numbers. This process is summarized below.

Figure 1: Basic Test-Driven Development Flow

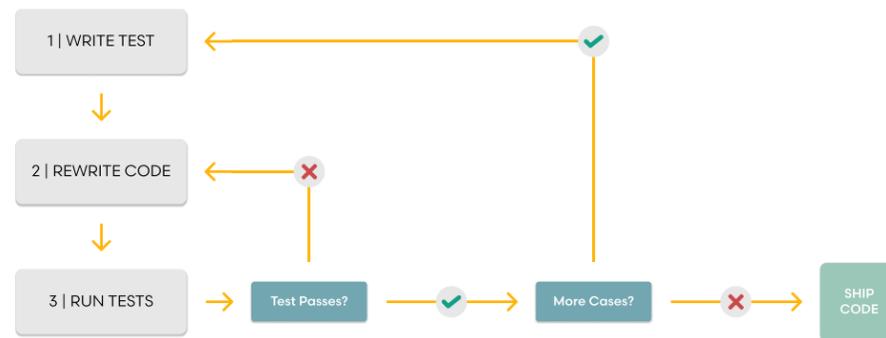


Photo by Patrick Shopfling on Unsplash

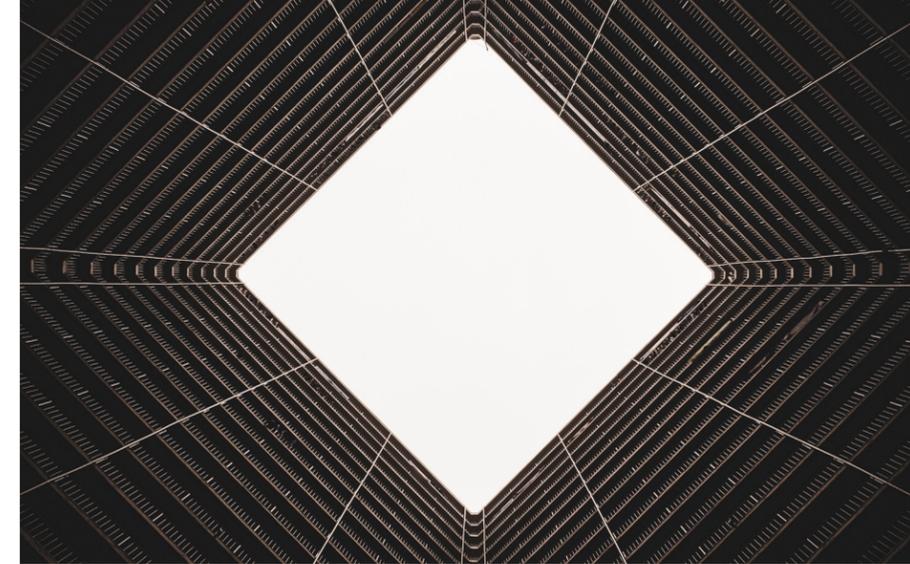
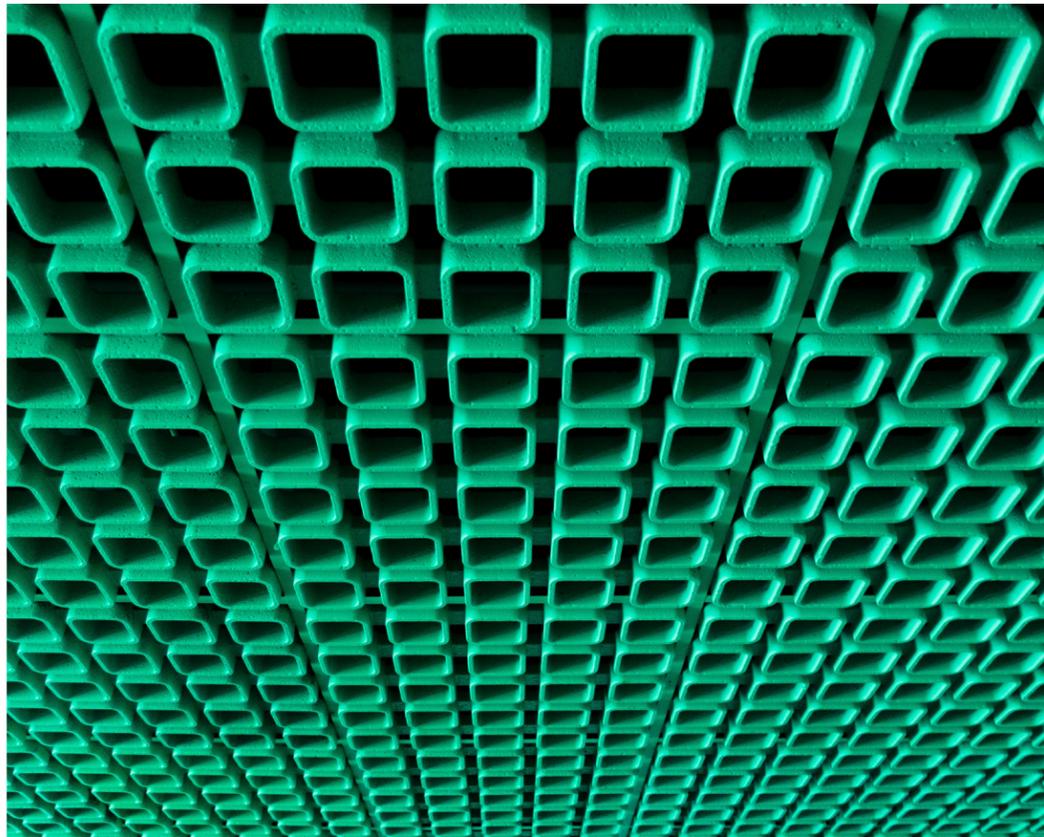


Photo by Laimannung on Unsplash

There are a few key aspects of the TDD process:

- ▶ While it can feel counterintuitive to begin with specific examples rather than writing code itself, this ensures that the requirements for the code are clear and agreed upon up front.
- ▶ It is important to think about what the code *should not* accept, in addition to what it *should* accept.
- ▶ Software testing requires creativity, and there is no single recipe for coming up with tests.
- ▶ Spending any amount of time brainstorming diverse tests like “911,” UK numbers, and “ABCDEF” before beginning the actual coding helps ensure that the resulting code is better written earlier in the process. Disagreements about how the code should behave can be discussed up front through the use of the chosen examples.
- ▶ Test suites grow over time and become increasingly robust. It can expedite the test development process to take tests from one setting and reuse them in another. For example, if the engineer working on the registration form later writes an iPhone app that also needs to accept phone numbers, the engineer could reuse many of the tests that were already produced, while adding new tests specific to the app.

Tests can be as sophisticated and creative as necessary for the given situation. A ride-hailing company like Uber or Lyft might have tests to ensure that a passenger is never matched with multiple cars at once; Google or Apple may want to ensure that driving routes they suggest in their maps never vary in duration by more than one hour; or an e-commerce company like Amazon may have tests to ensure that shipping fees are correctly calculated based on different geographic locations. Once tests are written and deployed, engineers can be confident that they are not introducing certain types of bugs when making changes or adding new features.

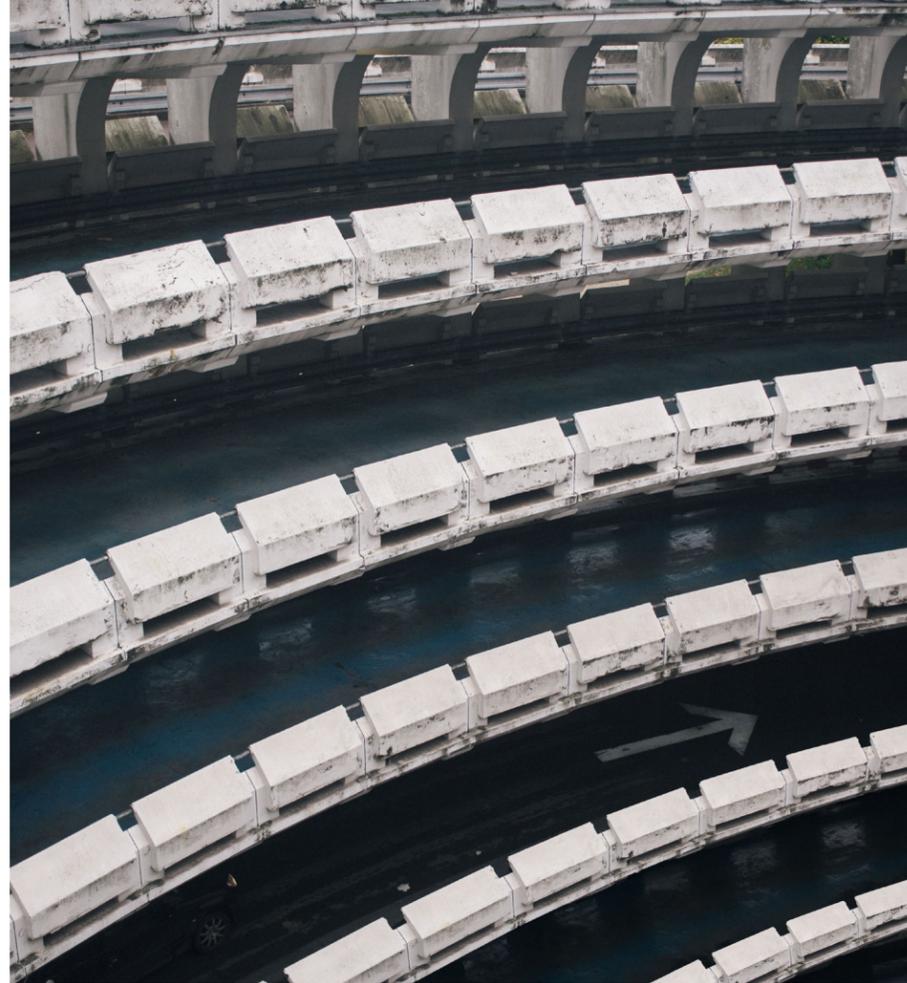


Photo by Etienne Boulanger on Unsplash

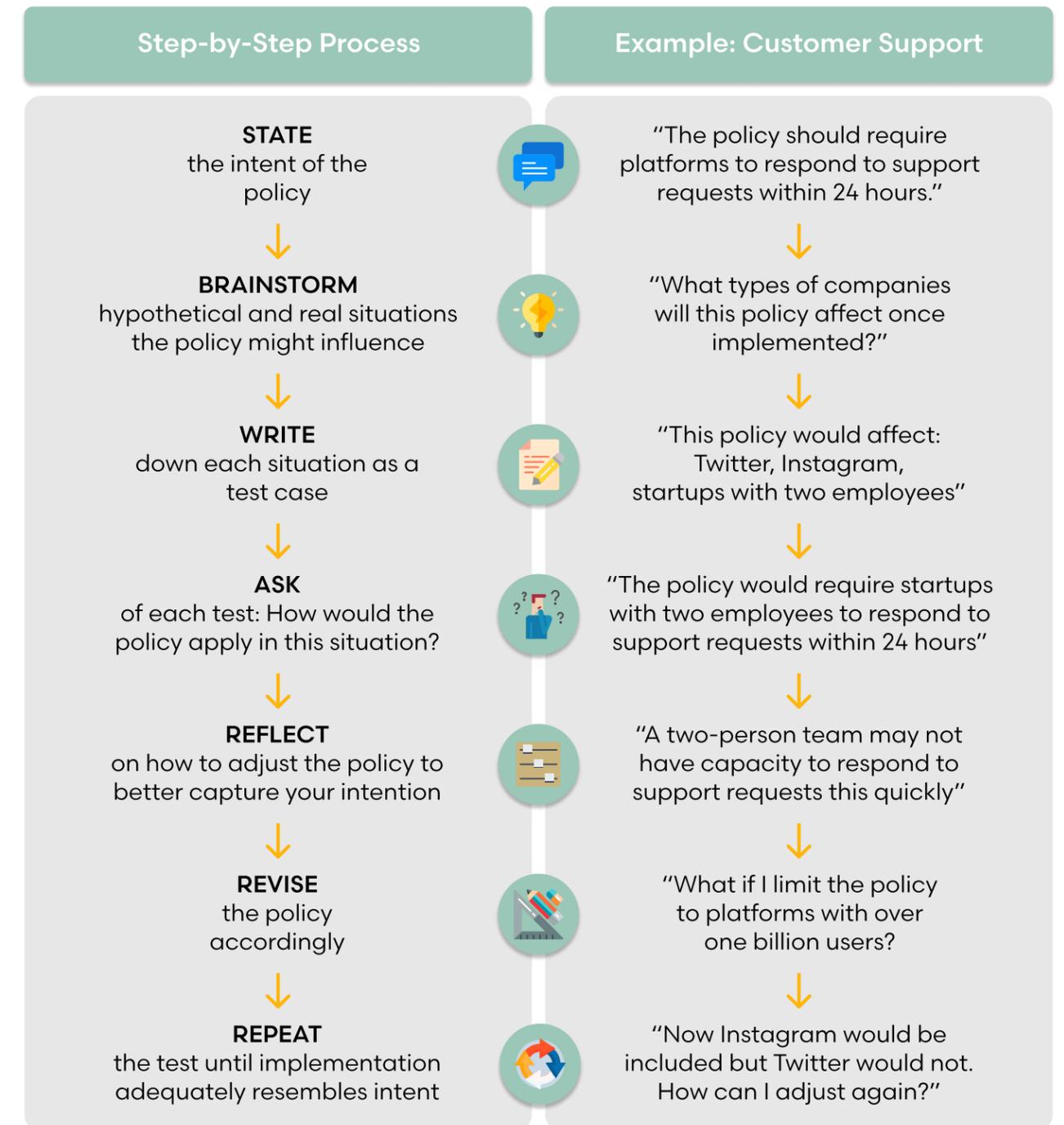
Test-Driven Development for Policymaking

These concepts from software testing can be adapted to the policy context as follows:

- ▶ A “test” is a snippet of text or other content in plain English that describes a given situation or example the policy is (or is not) supposed to address.
- ▶ The “code” being tested is the policy language being developed. This may be formal legal or legislative language or more informal language used earlier in the policymaking process.
- ▶ “Running the tests” involves manually looking through a collection of tests (a “test suite”) and comparing it to the policy language to see if the current version of the policy satisfies the requirements codified by the tests.

The iterative process of TDD is the same when applied to technology policy: policymakers should begin by brainstorming and agreeing on tests that indicate what the policy should or should not do, and then crafting policy language to satisfy the agreed-upon requirements.

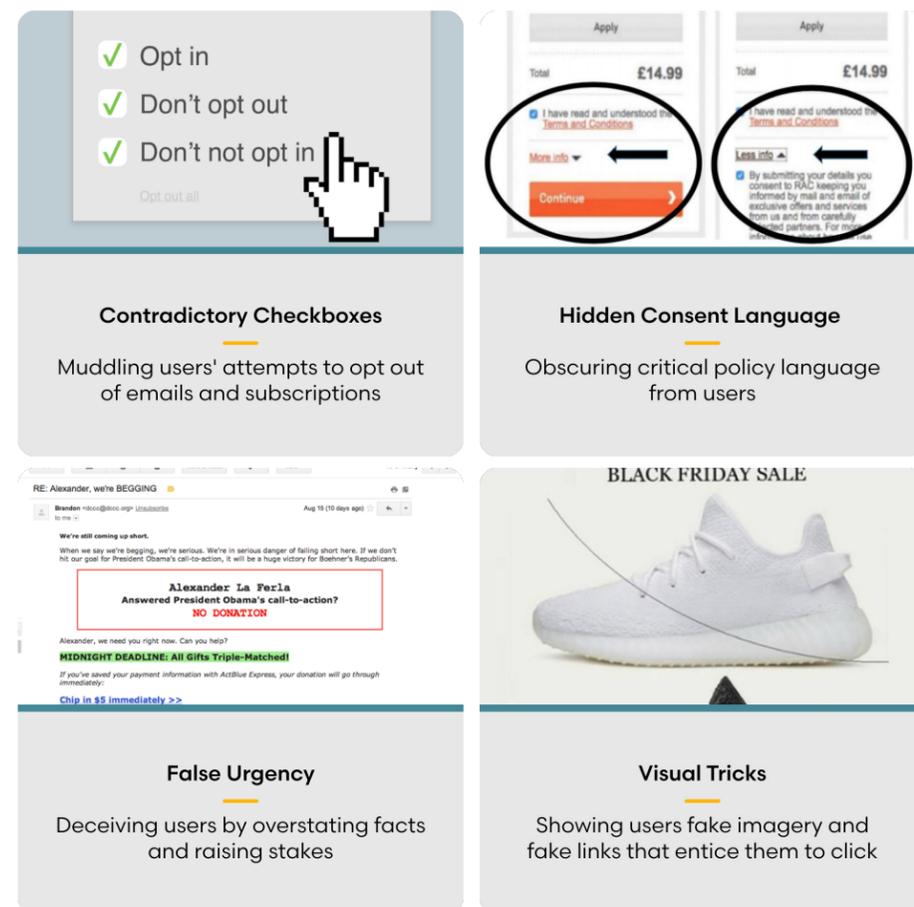
Figure 2: Test-Driven Development for Policy - A Step-By-Step Guide



Example: Dark Patterns. “Dark patterns” are manipulative designs used in websites, ads, and other software that attempt to deceive users into clicking on, buying, or signing up for something that they don’t want. The DETOUR Act⁷ was introduced in the 116th Congress to try to ban the use of dark patterns by so-called “large online operators,” websites like Facebook or Twitter that have more than 100 million monthly active users.⁸

Imagine a staff member in Congress is seeking to draft new legislation similar to the DETOUR Act to ban the use of dark patterns. The first set of tests the staff member would devise are relatively obvious: they are specific examples of both dark patterns that the bill should ban and legitimate marketing or design practices that the staff member wants to ensure the bill does not ban (see Figure 3). Even these few simple examples bring clarity to the goal of the legislation and how the policy language should be drafted to achieve the goals of the Act while not being so broad as to prevent legitimate marketing tactics, such as standard banner ads.

Figure 3: Dark Pattern Examples



In brainstorming tests, it can help to conduct research or consult with domain experts to help generate tests for new policies. Incorporating this expertise into the policymaking process is a key benefit of TDD. For example, the Princeton Web Transparency and Accountability Project has completed research on dark patterns and hosts a website with dozens of examples of dark patterns already organized into categories.⁹ These examples, or some selected subset, could be incorporated as part of the test suite.

Benefits. Using TDD for tech policymaking has several key benefits:

- ▶ TDD helps ensure that policymakers agree on concrete examples before crafting general language.
- ▶ TDD enables more experts to participate in forging policy that affects technology, as it is easier for technologists and domain experts to contribute and analyze tests than to engage with legalistic policy language.
- ▶ Policymakers can reuse and share test cases over time, adding efficiency to the process of drafting new laws and regulations.
- ▶ TDD does not require the use of any specialized tools or code.
- ▶ A single member of the policy team can begin using TDD effectively without requiring participation from the entire team. The practice will likely spread among the rest of the team as it proves useful.
- ▶ TDD can be used flexibly. A policy team can decide to use TDD for only a part of a given policy, meaning that it can help improve policy for that subset without limiting how the rest of the policy is put together.
- ▶ As in software, TDD can support the development of more robust policy-making even if the test suite is not perfectly comprehensive. For instance, in the phone number example above, the engineer may fail to consider whether users should be able to sign up with toll-free numbers. Even if the form inadvertently accepts such numbers, it will still be a better form than if the other tests had not been used.
- ▶ Clearly articulated tests can help chart direction for policy pilots or other evaluation methodologies that help ensure the policy’s implementation is effective.

How to write tests. As with software, writing tests for policymaking requires creativity in brainstorming, and there is no simple recipe to follow. However, there are some guidelines and prompts that policymakers can follow. Below, we list a set of ten “test categories,” along with conceptual questions that a policymaker can ask herself to arrive at sample tests.

Figure 4: Examples of Tests

Test Category	Questions	Test Examples
Numeric Thresholds	<ul style="list-style-type: none"> ▶ How does the policy change as numbers in the policy are adjusted? ▶ What happens to entities that pass in/out of the threshold? 	<ul style="list-style-type: none"> ▶ Covering online platforms with over 1M vs 100M vs 1B active users
Subcategory	<ul style="list-style-type: none"> ▶ What are all the distinct-subcategories of a category referenced in the policy? 	<ul style="list-style-type: none"> ▶ For content moderation policy, nudity could include pornography, historical photos, photo-journalism, nudes in art
Valid Practices	<ul style="list-style-type: none"> ▶ If some behavior is being banned, what is some similar but allowed behavior? 	<ul style="list-style-type: none"> ▶ Consider valid examples of online marketing or benign A/B testing in addition to dark patterns
Extreme Scenario	<ul style="list-style-type: none"> ▶ What are the failure modes of the policy under extreme or worst case scenarios? 	<ul style="list-style-type: none"> ▶ Making it impossible to treat any political group differently on platform makes it impossible to ban Nazis
Jargon	<ul style="list-style-type: none"> ▶ How do different people (engineers, domain experts, laypeople) interpret special jargon in the policy? 	<ul style="list-style-type: none"> ▶ DETOUR Act includes term “behavioral experiment”, which could be interpreted differently by engineers, users, social scientists
Excluded Entities	<ul style="list-style-type: none"> ▶ Are there entities included/excluded who do/don’t do the behavior in question? 	<ul style="list-style-type: none"> ▶ Google doesn’t use “fake hair” dark pattern ▶ Ticketmaster uses dark patterns but isn’t included
Stakeholder Impact	<ul style="list-style-type: none"> ▶ Who are all the entities affected by this policy, especially those not directly referenced? 	<ul style="list-style-type: none"> ▶ Hackers vs security researchers ▶ Malicious academic (Cambridge Analytica) vs benign academic
Business Incentives	<ul style="list-style-type: none"> ▶ What new incentives emerge for existing businesses? ▶ What new business models might spring from this policy? 	<ul style="list-style-type: none"> ▶ Requiring both large and small companies to implement the same privacy features would inhibit small company growth
Demographics	<ul style="list-style-type: none"> ▶ Does the policy make sense as you vary the attributes of the people/ companies covered? 	<ul style="list-style-type: none"> ▶ Demographics (age, gender, income, location), access to internet ▶ Market cap, funding stage, sector
Lawsuits	<ul style="list-style-type: none"> ▶ What potential litigation based on this language can you anticipate? 	<ul style="list-style-type: none"> ▶ ACLU sued state of Arizona over law banning ‘revenge porn’ that inadvertently banned use of historical and newsworthy images

Policymakers can come up with their own test categories, and through experience, they will get a sense of the types of tests and test categories that tend to be most helpful and relevant to particular tech policy areas. We recommend beginning with the most concrete test categories, such as Numeric Thresholds, Subcategory, Excluded Entities, and Jargon, as it is easiest to envision how to use them.

Standard office software, like a shared spreadsheet or document, can be used to store the tests. The team member responsible for running the tests can simply open the test suite and the policy language and compare them side-by-side, annotating as needed (e.g., commenting that a particular sentence or passage may cause test 13 to fail). (See Appendix A for a sample worksheet designed to guide a technology policymaker through the TDD process.)

Team structure. While individuals can use TDD effectively, engaging a larger team can make the method even more powerful. Many strategies are available for structuring teams to test policies:

- ▶ **Red teaming.** Playing the role of devil’s advocate in the policymaking process, a “red team” is an independent group tasked with challenging a system, often adversarially, to improve its effectiveness. Some large software companies use red teams by having one team within the organization responsible for writing tests and another team responsible for writing code. The natural competition between the test writers and code writers improves the reliability of the finished product. Red teaming can be applied to policymaking by appointing different people to be policy writers and policy testers, and having the two teams iterate on their work until the policy and tests are both robust. Red teaming has already been used in other parts of the policymaking process, especially for security policy.¹⁰
- ▶ **Including outside expertise.** TDD makes it easy for technologists, domain experts, and committees concerned with an issue to analyze or contribute tests, rather than policy language. Tests are often easier for experts to parse, especially those who do not work in policy on a full-time basis. Allowing experts to comment on and analyze draft tests can increase buy-in and confidence from those most informed about an issue.
- ▶ **Engaging a wider range of voices.** Policies often have unintended consequences because the policymaking process does not include a sufficiently wide range of affected stakeholders. In the case of technology policy, this could include everyone from the rank-and-file engineers in charge of implementing a policy at a software company to vulnerable populations who may be directly or indirectly affected by the policy. Using tests to invite such people to participate in the policy’s creation will help produce more robust policy.

Conclusion

Test-driven development can be adapted from the field of software engineering to make the tech policymaking process more robust. Tests written as snippets of text can help align policymakers around concrete examples that proposed technology legislation should and should not influence, and can expand the expertise involved in crafting new tech policy by engaging experts from industry and academia. Furthermore, tests can be recycled, shared, and extended over time as tech policy matures. The flexibility of test-driven development principles will improve policymakers' foresight and ease the effective implementation of new technology policy.

Photo by Daryan Shamkhali on Unsplash



Photo by Scott Webb on Unsplash

About the Authors

Neal Parikh is Co-Founder and former Chief Technology Officer of SevenFifty, a technology company based in New York. He holds a Ph.D. in computer science from Stanford University, focused on artificial intelligence and optimization. He previously taught machine learning at Cornell Tech and worked at Goldman Sachs in New York.

Ginny Fahs is a software engineer and social entrepreneur based in San Francisco. She is the Co-Founder and Executive Director of #MovingForward, a global social movement that has inspired over 100 venture capital firms to write external harassment and discrimination policies. Formerly she worked as a software engineer at Uber. She holds a degree in American History & Literature from Harvard University.

Brandie Nonnecke is Founding Director of the CITRIS Policy Lab and Co-Director of the CITRIS Tech for Social Good Program at the Center for Information Technology Research in the Interest of Society (CITRIS) and the Banatao Institute, headquartered at UC Berkeley. She is a Fellow at the World Economic Forum, where she serves on the Council on the Future of the Digital Economy and Society.

Acknowledgements

We would like to thank Chris Riley, Joe Hall, Betsy Cooper, Maitreyi Sistla, Ryan Olson, Cori Zarek, Ryan Calo, Zvika Krieger, and Nicole Tisdale for feedback and assistance with this project.





**ASPEN TECH
POLICY HUB**

 THE ASPEN INSTITUTE