The Budget That Cannot Be Overspent: From Governance Notaries to Governance Mechanisms

Şen Ferhat
Independent researcher
sen@kycel.com

July 2026, working draft v0.1

Abstract

Most on-chain governance systems are *notaries*: they record decisions but do not enforce them. An approved proposal emits an event, and execution is left to whoever operates the surrounding infrastructure. We present STOA, a framework that models institutions, states, companies, cooperatives, federations, as compositions of 25 domain-neutral smart-contract primitives (registries, assemblies, treasuries, budgets, timelocks, charters, work orders, arbitration), realized by ∼30 Solidity contracts totalling 2.8 kLOC, where the visual process map, the deployment manifest, and the running system all derive from a single graph. We contribute: (1) the *notary/mechanism* distinction, operationalized as an honesty taxonomy that labels every component either *self-enforcing*, its rule physically cannot be violated on-chain, or *attested*, a notarized claim about off-chain reality; 16 of our 25 primitives are self-enforcing, and the label is carried into the user-facing design surface and the deployment manifest; (2) an *actuation loop* in which an assembly vote compels the voted on-chain action through an appeal-window timelock, combined with a deploy-time *governance handover* after which the deployer provably loses control of the system it created; and (3) *budget-as-code*: appropriations approved by vote are enforced by the treasury itself as a spend policy consulted on every outflow, making overspending a transaction revert rather than a rule violation, including for the treasury's own owner. We demonstrate all three in a reproducible case study in which every bypass attempt by the deployer reverts, while the identical action succeeds when, and only when, it travels the constitutional path: proposal, vote, quorum, timelock, execution. We close with an honest account of what cannot be coded: identity, oracles, secrecy, and legal recognition.

1 Introduction

Smart-contract governance has been possible in principle since programmable blockchains appeared [1, 2], and "code is law" has served as both promise and warning for longer [3, 4, 5]. In practice, however, the overwhelming majority of deployed "governance" consists of vote *recording*: a proposal is text, approval is an event log, and whatever happens next, a payment, a role change, a parameter update, is performed by an administrator, a bot, or a multisig that could equally well have acted without the vote. We call such systems *governance notaries*. A notary is useful: it produces a tamper-evident history of what was decided. But it does not bind decisions to consequences. The decision and its enforcement remain separated by a layer

of trusted human or operational discretion, which is precisely the layer institutions have always struggled to constrain.

This paper is about closing that gap for a general class of institutions, and about being honest where it cannot be closed. We describe STOA, a framework in which an institution is specified as a graph of *contracts and wires*: every node of the design is one deployable Solidity contract drawn from a small domain-neutral primitive library, and every wire is an event-to-method binding between two of them. The same graph is simultaneously (a) the visual process map a designer edits, (b) the deployment manifest, and (c) after deployment, the live system, there is no separate "model" that could drift from what actually runs.

Within this framework we develop three contributions:

C1, The notary/mechanism distinction, made operational. We distinguish components whose rules are *self-enforcing* (⧫): the rule is checked in contract code on the state the contract itself controls, so violating it is not forbidden but *impossible*, the violating transaction reverts; from components that are *attested* (◊): the contract notarizes a claim about off-chain reality ("the invoice was paid by bank transfer", "the delivered work is acceptable") that some designated party asserts. The distinction is not a footnote; it is carried as a first-class field on every primitive, surfaced in the design tool and in the deployment manifest, so a user always knows which of their institution's rules can be broken and which cannot. Of the 25 primitives in the current library, 16 are self-enforcing and 9 are attested. Several contracts deliberately contain both halves in one object, which makes the boundary concrete (§3.3).

C2, An actuation loop with governance handover. An assembly primitive accepts proposals that carry on-chain *actions* (target addresses plus calldata). When a proposal passes, under a quorum rule, a threshold in basis points, and optional per-category *tiered* thresholds so that existential decisions demand supermajorities, the approved actions are queued into a timelock whose delay functions as an *appeal window*, after which anyone may execute them. A wired arbiter can suspend a queued operation while a dispute is heard, then resume or cancel it by ruling: the appeal window has teeth. Crucially, the deployment pipeline ends with a *governance handover*: administrative rights over every chip are transferred from the deployer to a governing chip (typically the timelock fed by the assembly), and the resulting control graph is recorded in the manifest. After handover the system is sovereign over itself: we show a deployer whose every direct intervention reverts (§6).

C3, Budget-as-code. We introduce a budget primitive that is not a governance-side check but a *treasury-side* one: the treasury consults the budget on *every* outflow path, including those initiated by its own owner. An assembly-approved appropriation is therefore not an instruction to a finance department; it is the physical outer bound of what the treasury can emit, per category and per fiscal period. A spend without a live appropriation line, or beyond a line's remainder, reverts. To our knowledge this inverts the usual arrangement, in the standard governor pattern the governance contract gates *its own* actions, while the treasury remains spendable by its operators; here the treasury itself refuses. This is the single clearest demonstration of what a coded institution can do that a paper institution cannot: *a government that cannot overspend its budget.*

We evaluate these mechanisms in a reproducible case study (§6): a minimal state, registry, senate, timelock, treasury, budget, is deployed and handed over; six bypass attempts by the

deployer and by unbudgeted spenders all revert with the expected reasons; the same appropriation then succeeds through the constitutional path. The framework, contracts, tests, and the case-study systems are available as a working artifact.[1]

We are equally explicit about limits (§7). Everything above holds *relative to the substrate*: on a chain the operator controls, "the contract guarantees X" collapses to "the operator promises X", so we position single-operator deployments honestly as *simulation mode*. Votes in the current assembly are public, one-address-one-vote presumes an identity layer that does not yet exist (a keypair is not a person), and nothing here confers legal recognition. The taxonomy of C1 is, in part, the discipline of saying this precisely.

2 Related Work

DAO frameworks and the governor pattern. Executable proposals are established practice in token governance: Compound-style governors and their OpenZeppelin generalization let a proposal carry targets and calldata, queue through a timelock, and execute on-chain [6]. Aragon and successors package DAO deployment and permission management [7], and the DAO design space is well surveyed [8, 9, 10, 11]. STOA differs in three ways. First, scope: the governor pattern governs a protocol; STOA's primitive library aims at *institutions* in general, its vocabulary includes tax offices, charters, budgets, work-order desks, land registries, and arbitration, not only voting and treasuries. Second, the budget inversion of C3: in governor deployments the timelock typically *owns* the treasury, so anything the timelock executes can spend; in STOA the treasury itself enforces appropriations against *all* callers, which is what makes "cannot overspend" literal. Third, the honesty taxonomy of C1 and the recorded control graph of C2 have, to our knowledge, no counterpart: existing frameworks do not systematically distinguish which of their guarantees are physical and which are attestations, nor treat deployer-disempowerment as a first-class, manifest-recorded deployment step. Adjacent mechanisms exist at the protocol layer: Tezos bakes self-amendment into its own upgrade process, Aragon Court and Kleros arbitrate disputes through staked jurors, and MakerDAO's emergency shutdown wires a last-resort consequence directly into the protocol. Each governs its own protocol's parameters; none aims at institutions in general, and in none does the treasury itself refuse its operators.

Governance tooling for communities. PolicyKit lets online communities encode governance procedures in code executed against platform APIs [12], and a broader movement studies member-governed online spaces [13]. These systems govern mutable platforms and rely on platform cooperation; STOA targets the value-bearing, adversarial end of the spectrum where enforcement must be self-executing.

Rules as code and computational law. Government initiatives explore publishing legislation in machine-consumable form [14], and computational-law scholarship examines blockchain's "lex cryptographica" [5]. STOA can be read as the constructive complement: not translating existing law into code, but asking what institutions look like when their operative rules are code from the start, while marking, per component, where code ends and attestation begins.

[1] Artifact repository and reproduction instructions to accompany the submission; the case study replays via the framework's contract test suite.

Institutional analysis. Our primitive vocabulary is a software answer to a question institutional economists have long asked in the abstract: what are institutions made of? Ostrom's design principles for commons governance, monitoring, graduated sanctions, nested enterprises [15], and the Crawford–Ostrom grammar of institutions [16] decompose institutions into typed, composable elements. STOA's library is deliberately congruent: groups, authorities, assemblies, obligations, and sanctions as composable contracts, with federation as nesting. We return to this mapping as future work; the oracle boundary we formalize as $\Diamond$ components echoes the known blockchain oracle problem [17].

3 The Framework: Institutions as Contracts and Wires

3.1 One graph, three readings

A STOA system is a graph. Nodes are *chips*: instances of primitives from the core library, each with a display label, a configuration, and typed *ports*, output ports correspond to contract events, input ports to contract methods. Wires connect an event port to a method port, optionally with a payload mapping. The same graph is read three ways: as the *process map* the designer edits and animates; as the *deployment manifest* (which contracts to deploy with which constructor arguments, then which wiring calls to make); and, after deployment, as the address book of the *live system*. Templates, 23 in the current distribution, from a village federation and a procurement process to a software company and two contrasting nation-states, are simply stored graphs, and the framework's design rule is that the core stays domain-neutral: domain content ships as removable packs, never as branches in core code.

3.2 The primitive library

The current library comprises 25 primitives realized by ∼30 Solidity contracts (2,808 LOC). Table 1 shows a representative subset with enforcement classes. The library is small by design: institutions are expressed as *compositions*, and a capability that cannot be expressed as a composition is the signal to extend the library, not to hardcode a system.

3.3 The honesty taxonomy

Every primitive carries an `enforcement` field: *self-enforcing* or *attested*. The field is not documentation; it is rendered as a badge in the design tool's technical view and written into the deployment manifest, so the distinction survives from design to operations. The point is product integrity: for a real organization, knowing *which rules can be broken* is the product.

Two contracts illustrate the boundary sharply because they contain both halves. The redemption desk burns scrip at the moment of a redemption request, supply shrinks irrevocably, a self-enforcing step, while the corresponding bank transfer is a tracked, *attested* step a finance officer later marks paid. The invoice book settles on-chain payments directly into the treasury (self-enforcing) but accepts an officer's attestation for payments that arrived by bank (attested). The taxonomy forces the designer to see exactly where their institution stops being physics and starts being trust, and it pre-empts the standard oracle objection [17] by making the oracle-dependent surface explicit and enumerable rather than implicit.

Primitive (contract)	Institutional role	Class
Registry (`StoaGroup`)	membership, roles, eligibility	⧫
Assembly (`StoaAssembly`)	proposals, quorum, tiered votes	⧫
Timelock (`StoaTimelock`)	appeal window, suspend/resume	⧫
Treasury (`StoaAccount`)	balances, conservation, spend policy	⧫
Budget (`StoaBudget`)	appropriations per category/period	⧫
Currency (`StoaToken`)	ERC-20 system money, mint policy	⧫
Flow (`StoaFlow`)	value movement between accounts	⧫
Charter registry (`StoaCharterRegistry`)	permissionless entity register	⧫
Tax office (`StoaTaxCollector`)	registry-scoped remittance	⧫
Work-order desk (`StoaWorkOrder`)	bounty custody, deadline refunds	⧫
Invoice book (`StoaInvoice`)	receivables; on-chain settlement	mixed
Redemption desk (`StoaRedemptionDesk`)	scrip exit; bank leg	mixed
Arbitration (`StoaArbitration`)	cases and rulings	◊
Oracle (`StoaOracle`)	external facts, single reporter	◊
Record (`StoaRecord`)	certificates, registries	◊

Table 1: Representative primitives. ⧫ = self-enforcing (rule checked on-chain against state the contract controls; violation reverts). ◊ = attested (notarized claim about off-chain reality). "Mixed" contracts contain both halves explicitly. Overall: 16 of 25 primitives self-enforcing, 9 attested.

4 The Actuation Loop

4.1 Executable proposals

The assembly accepts two kinds of proposals. A text proposal is a title and body, a notarized intention. An *actionable* proposal additionally carries parallel arrays of targets and calldata:

```
createProposalWithActions(title, body, votingSeconds,
                          targets, calldatas, category)
```

Voting is one-address-one-vote, gated by a pluggable membership registry (any contract exposing `isMember(address)`), with ballots yes/no/abstain and real-time voting windows. Finalization checks, in order: a quorum rule ($cast \cdot d \geq eligible \cdot n$ for a configured n/d), then an approval threshold in basis points over yes+no votes. The threshold is *tiered by category*: a category such as `existential` can be bound to 7500 bps, so a 60% majority that passes an ordinary measure fails a dissolution, constitutional rigidity expressed in one mapping. On approval, the assembly queues the proposal's actions into the wired timelock and emits `ProposalQueued`; text proposals queue nothing. The decision is now on rails: nothing further depends on any operator choosing to comply.

4.2 The appeal window has teeth

The timelock holds queued operations for a configured real-time delay, an *appeal window*, after which *anyone* may execute them; execution performs the calls and bubbles inner revert reasons. During the window, the operation can be cancelled by the admin or proposer; and a wired *arbiter* (a court chip) can `suspend` it, a suspended operation cannot execute even after its window lapses, then `resume` or cancel it by ruling. This is deliberately the seed of a judiciary: filing a case freezes execution, and the ruling either releases or annuls the act (§8).

4.3 Governance handover: the system becomes sovereign over itself

Every primitive is deployed with its deployer as administrator, unavoidable at construction, and the root of what we consider the deepest integrity flaw in deployed governance: the unaccountable superuser. A mechanism whose constitution can be bypassed by its deployer is a demo. STOA therefore treats *disempowerment as a deployment step*: after wiring, the pipeline transfers every chip's administrative rights to a designated governing chip, in the reference configuration, the timelock, whose only proposer is the assembly, and records the resulting *control graph* (who administers what) in the manifest the user sees. From that transaction forward, changing the budget, rewiring the timelock, or granting treasury operators is possible only as the output of a passed vote. The deployer keeps exactly the rights of any other member: proposing and voting.

5 Budget-as-Code

The treasury primitive maintains a conserved balance with two outflow methods (withdraw to an address; transfer to a peer account), an operator set, and an optional *spend policy*. When a policy is set, *every* outflow path first calls

```
authorizeSpend(operator, to, amount)
```

on the policy contract, which must revert to deny, and this check runs before balance accounting and regardless of caller, *including the treasury's own owner*. The budget primitive implements this interface as an appropriations ledger: named lines per category, per fiscal period; spenders (flow chips, officers) are assigned to categories; a spend with no line reverts `No appropriation line`, a spend beyond a line's remainder reverts `Appropriation exceeded`; each authorized spend increments the line atomically. Fiscal periods are explicit: a new period starts *empty*, so spending authority lapses by default until re-appropriated, the on-chain equivalent of an appropriations act expiring.

Two design choices matter. First, the enforcement point is the treasury, not the governance stack: the standard governor arrangement gates what *governance* executes, but leaves the treasury obedient to its operators; placing the check inside the treasury's outflow path closes the operator/owner bypass, which is exactly the hole real institutions cannot close (an officer with signing authority *can* overspend a paper budget; here the transaction reverts). Second, the budget's administrator is intended to be the timelock (§4.3), so appropriation lines themselves change only by passed vote: the assembly literally *is* the appropriations committee, and its approved budget is not an instruction but the outer physical bound of expenditure. The treasury operates in a symbolic-ledger mode for simulation and, when bound to the system's ERC-20 currency, in a custody mode where the same policy gates real token movement, the budget check runs before any token leaves.

6 Case Study: A Minimal State That Cannot Be Overspent

We compose five chips: a treasury (1,000 units), a budget bound to it as spend policy, a timelock with a real-time appeal window, a senate (quorum 1/2, ordinary threshold 5,000 bps), and payment flows (Figure 1).

The deployment pipeline performs the handover of §4.3: budget admin → timelock, treasury owner → timelock, senate admin → timelock; the senate is the timelock's only proposer. The

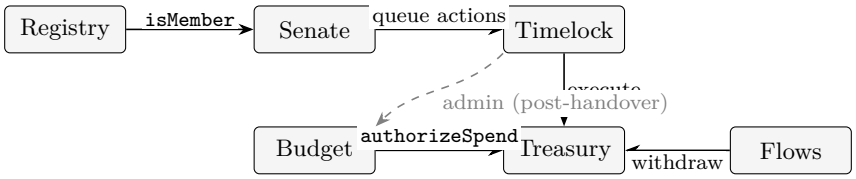

Figure 1: The case-study composition. The senate queues approved actions into the timelock; after the appeal window anyone may execute them against the treasury; the treasury consults the budget on every outflow, including flows and its own owner. After handover (dashed), administrative seats over budget, treasury, and senate are held by the timelock, whose only proposer is the senate.

Attempt	**Outcome**
Deployer sets an appropriation directly on the budget	revert: `Only admin`
Deployer grants itself operator rights on the treasury	revert: `Only owner`
Deployer re-points the senate at a new timelock	revert: `Only admin`
Unassigned flow fires a payment (no budget line)	revert: `No appropriation line`
Treasury owner path spends without a line ("owner grab")	revert: `No appropriation line`
Assigned flow spends 1 unit beyond its 100-unit line	revert: `Appropriation exceeded`
Execute a queued operation before the window lapses	revert: `Appeal window open`

Table 2: Bypass attempts in the case-study system after governance handover. Every non-constitutional path to expenditure or rule change reverts.

scenario replays deterministically in the framework's contract test suite; the same composition ships as a template and has been exercised as a live deployed instance of the framework's runtime.

6.1 Every bypass reverts

Table 2 lists the attempts. The deployer, the account that created every contract in the system, can no longer appropriate funds, seize operator rights, or rewire governance; unbudgeted spenders cannot move a single unit; budgeted spenders cannot exceed their line by one unit.

6.2 The constitutional path succeeds

A member then does what the deployer could not, lawfully: she submits an actionable proposal, "FY-1 schools budget", carrying `setAppropriation("schools", 500)` targeted at the budget chip, members vote, the proposal meets quorum and threshold, finalization queues the action into the timelock, the appeal window passes without suspension, and *anyone* executes the operation. The appropriation line now exists with 500 units, spendable by its assigned flows until exhausted or until the period rolls. Within one composition, the vote is the only path that works, and the vote *provably causes* the voted action: approval queues it, the window guards it, execution is permissionless, and enforcement afterward is the treasury's own refusal logic.

Under the taxonomy of §3.3, everything in this loop is ⧫: proposal admission, ballots, quorum arithmetic, tiered thresholds, queuing, the window, execution, appropriation accounting, and the spend gate. What remains ◊ is exactly what should be: *whether the schools were built* is a claim

about the world, and the system can only notarize it, a boundary the design surface shows rather than hides.

7 What Cannot Be Coded

The substrate bounds every guarantee. All claims above are relative to the chain: on a single-operator chain, "the contract guarantees X" means "the operator promises X". The framework's local runtime is therefore positioned as *simulation mode*, suitable for design, rehearsal, and research, and the guarantees become adversarially meaningful only on a substrate the participants do not individually control (a public network, or a consortium chain where each federated organization validates). This is a deployment decision the framework must surface, not bury.

A keypair is not a person. One-address-one-vote presumes identity. Key loss, key theft, guardianship, succession on death, and sybil resistance are unsolved in the current library, and no purely on-chain construction solves them; they require an identity layer with recovery and attestation, which imports trusted parties, and belongs on the $\Diamond$ side of the ledger. Relatedly, the assembly's eligible-population parameter is today set administratively (post-handover: by vote); wiring it to the registry is mechanical, but *who counts as a member* is not a question code can answer.

Votes are public. Ballots are readable on-chain forever. Coercion resistance and vote-buying resistance require secret ballots (commit–reveal at minimum), and confidential institutional records conflict with public state; we flag this as a known subtraction from what "election" means here.

No legal recognition. A passed proposal binds the chips; it does not bind a court, a bank, or a tax office. Bridging to legal reality requires jurisdiction-specific wrappers (association statutes, operating agreements) generated from the same graph, future work we consider essential rather than optional for real adoption.

8 Future Work

Amendment as typed patches. Institutions' most load-bearing rule is the rule for changing the rules. We are developing *patches*, subgraph diffs typed against the primitive vocabulary ("requires one anchor of kind `treasury`"), not against instances, as the unit of change: a live system adopts a patch by vote, so patch application *is* constitutional amendment, and the same format distributes improvements across all systems built on the vocabulary. An unverified patch is a constitutional coup vector, which makes verification (below) load-bearing.

Judgment as settlement. The timelock's suspend/resume/cancel arbiter hooks are the seed of a judiciary in which verdict and enforcement collapse into one atomic operation, damages transferred, licenses revoked, records annulled in the ruling transaction, under jurisdiction granted by consent at charter time. The concentration of power this implies (court capture) demands counterweights we intend to design as first-class patterns: judge terms, appellate layers, jurisdiction scoping.

Verifying institutional designs. The framework includes an LLM-assisted designer that proposes graphs over the primitive vocabulary. Generated constitutions demand more than structural validation: we plan a property layer checking institutional invariants, quorum reachability, no-single-drainer (including the deployer), an appeal path for every power, no orphaned administrator, graduated sanctions [15], run before deployment and at patch-adoption time. The library's size (2.8 kLOC) makes formal verification of the primitives themselves tractable.

Elections, offices, and the corporate spine. Representative structures (candidacy, terms, recall), obligations with default consequences, weighted voting over a share register, and secret ballots are the next primitives; each is a composition test for the vocabulary.

9 Conclusion

The distance between recording a decision and causing it is the distance between a notary and a mechanism. We have shown a small, domain-neutral primitive library in which that distance is closed for a meaningful class of institutional rules: a vote that provably causes the voted action, a deployer that provably cannot override the system it created, and a budget that provably cannot be overspent, with an honest, machine-carried boundary marking where enforcement ends and attestation begins. The bar we hold ourselves to, and propose for the field, is that the rules for changing all of the above should themselves be on-chain; typed patches adopted by vote are our next step toward it.

References

[1] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. `https://ethereum.org/en/whitepaper/`, 2014. Whitepaper.

[2] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. `https://ethereum.github.io/yellowpaper/paper.pdf`, 2014. Yellow paper.

[3] Lawrence Lessig. *Code and Other Laws of Cyberspace*. Basic Books, New York, 1999.

[4] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[5] Primavera De Filippi and Aaron Wright. *Blockchain and the Law: The Rule of Code*. Harvard University Press, Cambridge, MA, 2018.

[6] OpenZeppelin. Governance — OpenZeppelin contracts documentation. `https://docs.openzeppelin.com/contracts/governance`. Accessed July 2026.

[7] Luis Cuende and Jorge Izquierdo. Aragon network: A decentralized infrastructure for value exchange. `https://github.com/aragon/whitepaper`, 2017. Whitepaper.

[8] Shuai Wang, Wenwen Ding, Juanjuan Li, Yong Yuan, Liwei Ouyang, and Fei-Yue Wang. Decentralized autonomous organizations: Concept, model, and applications. *IEEE Transactions on Computational Social Systems*, 6(5):870–878, 2019.

[9] Samer Hassan and Primavera De Filippi. Decentralized autonomous organization. *Internet Policy Review*, 10(2), 2021.

[10] Youssef El Faqir, Javier Arroyo, and Samer Hassan. An overview of decentralized autonomous organizations on the blockchain. In *Proceedings of the 16th International Symposium on Open Collaboration (OpenSym 2020)*. ACM, 2020.

[11] Vitalik Buterin. DAOs, DACs, DAs and more: An incomplete terminology guide. `https://blog.ethereum.org/2014/05/06/daos-dacs-das-and-more-an-incomplete-terminology-guide`, 2014. Ethereum Foundation Blog.

[12] Amy X. Zhang, Grant Hugh, and Michael S. Bernstein. PolicyKit: Building governance in online communities. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*, pages 365–378. ACM, 2020.

[13] Nathan Schneider. *Governable Spaces: Democratic Design for Online Life*. University of California Press, Oakland, 2024.

[14] James Mohun and Alex Roberts. Cracking the code: Rulemaking for humans and machines. OECD Working Papers on Public Governance 42, OECD, 2020.

[15] Elinor Ostrom. *Governing the Commons: The Evolution of Institutions for Collective Action*. Cambridge University Press, Cambridge, 1990.

[16] Sue E. S. Crawford and Elinor Ostrom. A grammar of institutions. *American Political Science Review*, 89(3):582–600, 1995.

[17] Giulio Caldarelli. Understanding the blockchain oracle problem: A call for action. *Information*, 11(11):509, 2020.