SpacesOS

Christopher Robison

v0.1.0 – July 9th, 2026

Abstract

The operating system (OS) is the metaphor through which society understands the functional role, and therefore the consensus purpose, of computers. As the structure of the OS changes, so too does the metaphor and, by extension, society. The metaphor of the operating system can be understood by its capabilities and chief contextual usages. Historically, the OS has been likened to:

- **Governments** – in the mainframe era, the OS *regulated* its machine's scarce resources
- **Workshops** – in the unix era, the OS offered users a collection of basic *tools* (ed, cat, grep)
- **Offices** – in the personal computer era, the OS allowed users to be *productive* & *collaborative*
- **Town squares** – in the smart phone era, the OS *connected* people globally (social media)
- **Infrastructure** – in the cloud era, the OS *receded* into the background, functioning primarily as a *portal* to remote services

In the present *agentic* era, we propose that the new metaphor of the operating system is that of **space**. A person lives in their own space. They share a space with others. Spaces can be unique to a time or place. They can be private or public. Spaces are ubiquitously all around us. And above all, a space can be stratified — assembled from discrete, identifiable components — and therefore may be *ownable*.

A society where members own their computational environments with complete agency is one that prioritizes human culture above all else. It respects emergent beliefs, values, behaviors, and objects as the core drivers of civilization, rather than top-down directives issued by powerful, governing bodies. It also transforms the concept of an OS from a simple desktop to a complex build system capable of generating a variety of unique, digital environments & architectures. AI, often classified as a centralizing force, here may be run locally to achieve the inverse effect, functioning as a tool to give ordinary users the ability to assemble and inhabit intimately their own, diverse computational environments; their own sovereign *spaces*.

Large language models (LLMs) have naturalized computational literacy for the masses, albeit through abstraction. Consequently, previous paradigms have begun to quickly collapse, as users acquire an ever increasing capacity to prompt their machine (hardware) directly. Of all the artifacts from previous eras, which are heading towards obsolescence, none seems more consequential than the software provider.

IBM, Microsoft, Apple, Google, Amazon and all the various Linux distribution foundations have built operating systems for their consumers for nearly three-quarters of a century. We might define these as provider-procured operating systems (`p`OS) and suggest that a divergence has begun to occur whereby users are able to assemble their own bespoke systems from scratch using an agent-procured operating system (`a`OS). Indeed, the latter is the proposed design of SpacesOS (Spaces).

Spaces is a *declarative operating system* and *distributed package manager* (PM), anchored to public-private key pairs. The OS is natively equipped with first-class, interchangeable, local LLMs for system procurement, while the PM supply chain delivers publicly-verified, isolated dependencies and is therefore hyper-responsive to realtime continuous integrations. Configurations are lazily evaluated – assembled as needed – and thus easily reproducible. Everything is free & open source (FOSS).

Table of Contents

1 Problems

The provider-procured operating system cannot deliver environment ownership. This failure has two distinct causes:

- **Structural** — the *imperative architecture* on which pOS systems are built is incapable of producing the portable, reproducible artifact that environment ownership requires.
- **Organizational** — even when the underlying code is open and the architecture is sound, the existence of a *central, coordinating provider* creates a single point of leverage that adversaries, regulators, and misaligned commercial incentives can capture.

Additionally, there is an issue of **accessibility**. Both the structural and organizational problems require *complex, technical foundations* to be built in order to deliver true environment ownership. Such designs are generally inaccessible to the average user.

1.1 The Structural Problem

1.1.1 Imperative Configuration

An imperative operating system is configured through a sequence of commands. Each command modifies the state of the machine. The installation of a package, the activation of a service, the modification of a setting are each applied on top of the previous state, and the resulting environment exists only as the cumulative residue of every command that has been previously executed.

This residue cannot be inspected as a coherent object. It cannot be reproduced from a specification. It cannot be ported to another machine. There is no canonical representation of the environment apart from the machine itself. The environment is the machine, and the machine is the environment.

Example

If a user wants to configure an SSH server, they might run the following sequence of imperative commands:

```
apt install openssh-server
echo "Port 2222" >> /etc/ssh/sshd_config
echo "PasswordAuthentication no" >> /etc/ssh/sshd_config
systemctl enable sshd
systemctl restart sshd
```

Similar sequences must then be repeated for each and every package, service, and setting on the user's machine. The machine itself has no memory of prior commands. It is the responsibility and burden of the user to recollect and maintain them.

Consequences

Imperative configurations easily drift, conflict, accumulate undocumented modifications, and develop dependencies on conditions that exist nowhere except in the memory of the original administrator. When the machine fails, the environment dies or becomes corrupted along with it. When the machine is replaced, the environment must be rebuilt from scratch through another sequence of commands, which will produce a *similar* environment but never the *same* one. The architecture itself simply does not produce an ownable object.

This fragility pushes users toward enterprise-managed platforms whose complexity can only be maintained by a dedicated team of systems engineers. In exchange for convenience, users surrender control and agree to whatever terms-of-service may be offered. They then inhabit *borrowed* environments to which they can gain login-access, but which will almost certainly change, disappear, or become inaccessible at the custodian's discretion.

We refer to this systemic condition as **the shifting sands**: computing environments that cannot be reliably owned, preserved, or reproduced.

1.1.2 The Declarative Response

A declarative operating system inverts the relationship between configuration and machine. Instead of issuing sequential commands that modify state, the user writes a single configuration file that describes the desired environment in its entirety. The system then realizes that specification — installing packages, configuring services, resolving dependencies — by treating the specification as a single function whose inputs produce a deterministic output.

Here, the *specification* is the environment, not the machine. The machine is merely the substrate on which the specification is executed. This is the architectural basis of environment ownership. Without a declarative foundation, the environment has no independent existence. With one, it becomes a stable, first-class object — auditable, portable, reproducible, and defensible.

Example

If a user wanted to setup the same SSH server as before, they would simply open their single configuration file and add the following lines:

```
1   services.openssh.enable = true;
2   services.openssh.ports = [ 2222 ];
3   services.openssh.settings.PasswordAuthentication = false;
```

Anytime the owner wants to install or update additional packages, services, or settings, they only need to reopen the configuration file and edit the desired section. Then they simply command their computer to rebuild the system to meet the new specifications. The user inherits no burden of having to recount each and every modification. Instead the configuration, itself, becomes an explicit, ownable artifact that can be manipulated directly at the user's discretion.

Nix

Declarative systems are not new. Nix, the package manager and language on which Spaces is built, has implemented declarative configurations since 2003[1] with tremendous success. The `nixpkgs` monorepo is the largest public package repository in the world[2] containing more than 120,000 packages[3]. It is used to assemble its own NixOS Linux distribution, which has been used in production by CERN[4], the European Space Agency[5], Google[6], Shopify[7], and Mozilla[8]. Thus, it is reasonable to assume the structural problem is already solved in principle.

Implications

A declarative system has strong portability. As long as a user possesses a copy of their configuration file, they own their environment. A machine that loses its environment can recover it by re-evaluating the specification. Similarly, two machines that evaluate the same specification produce identical environments. An environment can be transferred between machines by transferring all or part of a specification. And a previous environment can even be restored by rolling back changes to a previous configuration.

It also offers strong buildability. Declarative systems evaluate their configurations lazily and represent packages as derivations with explicit, isolated dependencies. If two packages require different versions of the same dependency, both can be installed side by side in distinct store paths without conflict or additional intervention from the user. This isolation reduces dependency collisions and makes configurations easier to reproduce across compatible machines, provided their architecture, platform, and other declared inputs are the same.

1.2 The Organizational Problem

1.2.1 Provider Capture

A declarative foundation is necessary but not sufficient. NixOS itself is a declarative system and, while fully open source, still remains a ₚOS, because its packages, defaults, and updates are coordinated by a foundation. The foundation can be served with legal orders. It can be infiltrated. It can be pressured. It can be subverted by its own members or sponsors. The code is auditable, the license is permissive, the architecture is declarative — and still, a coordinating party exists, and that party is an attack surface.

1 https://nixos.org/research/
2 https://repology.org/repositories/graphs
3 https://github.com/NixOS/nixpkgs/blob/master/README.md
4 https://cds.cern.ch/record/2700235/files/10.1051_epjconf_201921405005.pdf
5 https://discourse.nixos.org/t/nixcon2023-nix-in-space/32934
6 https://firebase.studio/blog/article/nix-on-idx
7 https://shopify.engineering/shipit-presents-how-shopify-uses-nix
8 https://github.com/mozilla/release-services/tree/master/nix

This is true of every operating system in current use. Brazil has compelled providers to implement national identity verification at the operating system level[9]. Several jurisdictions in the United States, including California[10], are now considering similar mandates for age verification. The United Kingdom and the European Union have legislated against end-to-end encryption in ways that operating system providers are increasingly required to accommodate[11]. None of these mandates required the provider's consent. They required only the provider's *existence*.

We refer to this as **provider capture**. It is independent of the structural problem and persists even when the structural problem is solved. A perfectly declarative system maintained by a single, captureable provider remains a system that can be revoked, modified, or surveilled at the provider's compulsion.

1.2.2 The Distributed Response

The organizational problem is solved by eliminating the coordinating party. If the packages that compose an environment can be discovered, verified, and distributed without reference to any central provider, then there is no party left to capture. The environment thus becomes defensible not only structurally — through the declarative specification — but also organizationally, through the network that delivers its components: the distributed package manager.

Federated and Independent Repositories

There is indeed precedent for a distributed package manager. In 2025 the Linux Foundation launched the Federated and Independent Repositories (FAIR)[12] project in an effort to decentralize the Wordpress package supply chain, which currently runs approximately 42% of all online content management systems[13].

1.3 The Accessibility Problem

1.3.1 Terminal Illiteracy

While a declarative operating system and distributed package manager both solve the structural and organizational problems of environment ownership, they do not solve the problem of usability. These tools require at least a modest familiarity with the command line interfaces (CLIs), a skill which is not possessed by most of the general public. One narrow study suggests even a majority of system administrators prefer graphical user interfaces (GUIs) over CLIs[14].

9 https://www.planalto.gov.br/ccivil_03/_ato2023-2026/2025/lei/l15211.htm
10 https://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=202520260AB1043
11 https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A52025DC0148&qid=1747852533412
12 https://fair.pm/
13 https://w3techs.com/technologies/history_overview/content_management/all/y
14 https://arxiv.org/abs/1905.13582

We refer to the widespread avoidance of and inability to use CLIs as **terminal illiteracy**. It is self-reinforcing: if left unresolved, these tools fall further into irrelevance; if overcome, their usefulness grows as more people adapt them to their own needs to share with others. The ₐOS benefits from the network effects of users developing bespoke environments and actively sharing them with peers, which can be done as casually as sending a message. Greater collective literacy enables a greater experimental velocity and, therefore, greater diversity in digital culture[15].

1.3.2 The LLM Response

Locally trained and run FOSS LLMs are the great equalizer to resolve terminal illiteracy. A user who wants to install a package or service describes the module; the LLM modifies the configuration. A user who wants to share photos with friends describes the use case; the LLM assembles a space from the appropriate components. The user does not need to read or write the configuration file directly nor crawl the distributed package manager manually. They need only to express intent to an *agent command interface* (ACI).

The local model possesses three base categories of skills:

- *System-administrative assistance* — crawls the distributed PM, recommends modifications to system configurations, executes approved actions, and diagnoses problems.
- *Space control* — translates commands directly into operations that control installed packages and services, as well as manipulate local files and data.
- *Generative tooling* — produces the components a user requires from scratch, rather than installing pre-existing packages. Here, the distinction between *using* a system and *building* a system collapses.

This functionality is the engine of the agent-procured operating system. It is worth noting, however, that the word "agent" in ₐOS more broadly refers to a user's ability to express their own *agency* in being able to build their bespoke systems. As it happens, the best interface for doing this is with an LLM agent.

2 Spaces

The solutions to the three problems identified in §1 — (i) declarative configurations, (ii) a distributed software supply chain, and (iii) native agentic command interfacing — coalesce to produce a single, sovereign, ownable object: the *space*. A space is a computational environment that is configured declaratively, assembled without central authority, and commanded agentically.

15 https://vitalik.eth.limo/general/2025/12/17/societies.html

The term *space* is deliberately abstract. A user is invited to project onto it whatever concept makes sense for their use. Spaces are personal, communal, and infrastructural. They are as ubiquitous in digital life as they are in physical life. The vocabulary is intended to be portable across contexts — a person should be able to describe their digital existence in terms of the spaces they occupy, the spaces they share, and the spaces they are building.

2.1 Hierarchy

A user's primary environment is their *root space*. From the root space, additional *sub-spaces* may be instantiated — isolated, purpose-specific environments that exist alongside the root without interference. A user may run a coding space, a personal finance space, a research space, and a media space simultaneously, each with its own packages, services, agents, and data, each cryptographically bound to the same private key.

Spaces may be entered, exited, linked and nested. This structure allows the user to compartmentalize trust and isolate workloads while retaining the continuity of a single, fluid desktop environment. Spaces may be shared with collaborators and, subsequently, paused, archived, or destroyed.

2.2 Structure

SpacesOS runs directly on the hardware. It is the root *space* from which the user instantiates new *sub-spaces* using a kernel-based virtual machine (KVM-based) hypervisor. In its current construction, each sub-space is deployed as a microVM, a lightweight virtual machine with its own kernel that is isolated from the host. Such compartmentalization was originally pioneered by QubesOS[16] and Genode[17], among others. The main differentiation here is that each space is assembled from its own declarative configuration file.

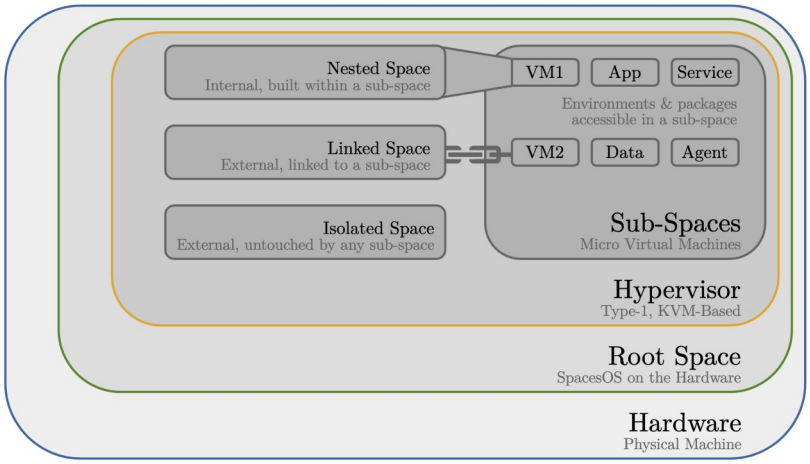

Figure 1: Hierarchy & Structure of Spaces

16 https://www.qubes-os.org/attachment/doc/arch-spec-0.3.pdf
17 https://genode.org/

Example

If a user wanted to create a new space for their family to privately share photos, they could generate a new configuration file — either on their own or with the aid of an LLM[18] — where all packages and network settings would be assembled from ~45 lines of code.

```
{ config, pkgs, ... }:
{ networking.hostName = "family-photos-space";

  # Store and display the family photo library with the Immich app.
  services.immich = {
    enable = true;
    host = "127.0.0.1";
    port = 2283;
    mediaLocation = "/srv/family-photos";
  };

  # Expose Immich through a web server.
  services.nginx = {
    enable = true;
    virtualHosts."photos.example.com" = {
      forceSSL = true;
      enableACME = true;
      locations."/" = {
        proxyPass = "http://127.0.0.1:2283";
        proxyWebsockets = true;
        extraConfig = ''
          # Only these family members' IP addresses may connect.
          allow 2001:db8:100::1; # Mom
          allow 2001:db8:100::2; # Dad
          allow 2001:db8:100::3; # Grandma
          allow 2001:db8:100::4; # Sibling
          allow 2001:db8:100::5; # Uncle
          deny all;
          client_max_body_size 50G;
          proxy_request_buffering off;
          proxy_read_timeout 600s;
          proxy_send_timeout 600s;
        '';
      };
    };
  };
  networking.firewall.allowedTCPPorts = [ 80 443 ];

  # Required for automatic HTTPS certificates.
  security.acme = {
    acceptTerms = true;
    defaults.email = "you@example.com";
  };
  system.stateVersion = "25.11";
}
```

18 In fact, this particular configuration example was indeed generated by an LLM.

2.3 Dependency & Package Isolation

One efficiency gained from using a declarative system in the context of spaces is that of package and dependency isolation. Declarative systems automatically resolve package dependencies into a closed, content-addressed graph[19]. Every package — together with the exact versions of every library, tool, and configuration it requires — is installed at a path derived from the cryptographic hash of its inputs. Two packages that depend on different versions of the same library coexist without conflict, because each version occupies a separate hashed path. The resulting system has no global dependency state to corrupt and no possibility of one package's installation interfering with another's.

This property extends naturally to spaces. When two spaces depend on the same package, the host stores it only once. Each microVM mounts that single store path from the host read-only and treats it as part of its own filesystem. The binary on disk is shared; the running process is not. When a package executes inside a sub-space, it runs in that sub-space's own memory, under its own kernel, behind the hypervisor's boundary. A process running inside one sub-space cannot write to the shared store, cannot observe the memory or runtime state of a process in another space, and cannot use the shared installation as a channel for communication. The fact that two sub-spaces execute the same binary grants neither of them any access to the other.

2.4 Data Isolation

While packages are isolated by the hypervisor, data must be isolated by the user. Data must be created, modified, shared, and revoked according to rules that reflect the user's intent, not the package's structure.

Spaces handle data through *data containers*, an abstraction inspired by Project Wildland[20]. A data container is a self-defined, signed unit of data, which can be attributed to its owner's public key and addressable by one or more user-defined paths. Containers are not bound to any specific storage backend. They can live on local disk, a home server, an external storage provider, or a peer's machine, and they can be migrated between backends without changing how the user references them.

Each container declares its own access rules. A user may grant read-only access to a friend, read-write access to a collaborator, or append-only access to an agent — each grant cryptographically bound to the recipient's public key. The same container can therefore appear in multiple spaces with different permission sets. A photo container shared with family in one space may also be shared, read-only, with an agent in a research space that has been asked to organize the collection.

19 https://discourse.nixos.org/t/nix-tree-interactively-browse-the-dependency-graph-of-your-nix-derivations/
20 https://golem.foundation/resources/documents/wildland-w2h.pdf

Data can be moved between spaces in two ways. It can be *linked* — exposed to a new space without being copied, with all access rules continuing to apply. Or it can be *duplicated* — reconstituted as a new container within the receiving space, governed by whatever rules the receiving space defines. Linking preserves the canonical version. Duplication creates a fork. Both are valid, and the user chooses according to the requirements of the workflow.

The hypervisor enforces the isolation between spaces, but the data container model enforces the isolation between *uses* of data within and across spaces. A package running inside a sub-space cannot read a container it has not been granted access to, even if the container is technically present on the same machine. Access is a cryptographic property of the container, not a property of the filesystem.

2.5 Private Key Access

The isolation properties described above may be enforced cryptographically. Spaces are able to be bound to private keys, where the most consequential operations on or between spaces may be executed with a signature to add an additional layer of security. Four operational examples, pertaining to spaces, that may be executed with a private key include:

- *Instantiation* — creating a new space can be done with a signature to bind ownership and admin authority to a space.
- *Data transfer* — linking or duplicating a container from one space to another may be authorized with a signature.
- *Access sharing* — granting a new user the ability to enter or modify a space is an act the owner may choose to sign.
- *Space merging* — merging two spaces together so that one is subsumed by another may require a signature.

The local cryptographic primitive used to root the key is not specified. Users should be able to use their preferred signature scheme or — in unique, likely local-exclusive use cases — design their system with no key at all. What matters architecturally is not the choice of primitive, but the principle: ownership of a space can be defined as a cryptographic property of an ownable artifact, rather than as a credential issued by a 3rd-party provider.

Keys are, however, entirely necessary when interacting with certain components of the distributed package manager.

3 Distributed Package Manager

The distributed package manager is composed of four elements:

1. a *package store* on each user's machine
2. a *peer transfer* protocol for moving packages between machines
3. a *builder network* that produces and attests to package builds publicly
4. a *system store* in which encrypted configurations are published for cross-machine reconstitution

3.1 Package Store

Every machine running Spaces maintains a local package store as described in §2.3. The store follows the architecture established by Nix[21] where each package is installed at a path derived from the cryptographic hash of its inputs, producing entries of the form `/nix/store/<hash>-<name>-<version>`.

The store is content-addressed rather than location-addressed. Thus, a package is identified by what it *is*, not by where it came from. This property makes it possible for the distributed package manager to run in a p2p manner. Two users who both build Firefox version X from the same specification produce bit-identical outputs at the same hashed path. A package retrieved from a friend, a stranger, or a build farm is verifiable against the hash without reference to its source. Trust in the package does not require trust in the channel that delivered it.

The package store is therefore both a local cache and a distributable artifact.

3.2 Peer Transfer

A user requesting a package first checks their own store. If the package is not present, they request it from peers. The peer transfer protocol operates in two modes.

In the *private mode*, a user requests packages directly from peers they trust — friends, family, colleagues, members of their clan. These transfers occur over an authenticated, encrypted channel, and neither party needs to obscure the contents of the request. A well-connected user, whose peers run similar configurations, will rarely need to leave the private mode.

In the *public mode*, a user requests packages from peers they do not know. This mode is necessary when no trusted peer holds the desired package, but it introduces a privacy concern: the responder learns which packages the requester is assembling, which can reveal information about the requester's environment, activity, and intent. The protocol provides three mechanisms for preserving privacy in this mode.

21 https://nix.dev/manual/nix/2.24/store/types/local-store

3.2.1 Randomized Package Store

A user may opt to participate in a *randomized package store*. Their machine downloads, stores, and serves packages on a randomized rotation, independent of which packages the user actually needs. From the perspective of any observer, the user appears to be requesting and serving a continuous stream of unrelated packages. The packages the user actually requires arrive in this stream alongside everything else, indistinguishable from the cover traffic.

The randomized store is implemented as a dedicated sub-space. Random packages are downloaded and stored within that sub-space in isolation from the user's working environments. When the user requires a specific package for use in another space, they transfer it from the randomized store space into the relevant space. The cost of this approach is bandwidth and storage overhead; the benefit is that an observer cannot infer the user's actual configuration from their network traffic.

3.2.2 Oblivious Transfer

For targeted requests, the protocol may utilize *oblivious transfer*, a cryptographic primitive in which a receiver retrieves one of n items from a sender without the sender learning which item was retrieved, and without the receiver learning anything about the other $n - 1$ items. Applied to package retrieval, this allows a user to obtain a specific package from a peer without revealing which package was requested.

Oblivious transfer is more bandwidth-efficient than the randomized store for users with specific, time-sensitive needs. It is also more cryptographically demanding. The two mechanisms are complementary: the randomized store provides ambient privacy at a constant cost, while oblivious transfer provides targeted privacy on demand. A user may employ either or both, depending on the threat model and the urgency of the request.

3.2.3 Onion Routing

The randomized package store and oblivious transfer both protect the *content* of a request — what package a user retrieves. Neither protects its *origin*. A responding peer, or an observer watching the network, can still learn the requester's address and, over time, correlate a machine with the packages it seeks.

Onion routing conceals the origin. Modeled on Tor, it routes a request through a circuit of relays, wrapping it in successive layers of encryption — one per relay. Each relay decrypts only its own layer, learning the next hop and nothing more. The first relay knows the requester but not the package; the last knows the responding peer but not the requester. No single relay can link the two, and deanonymizing the requester requires controlling both ends of the circuit at once. The relays are other network participants, contributing bandwidth as peers elsewhere contribute storage.

Onion routing composes with the other two mechanisms — concealing the origin of cover traffic, or of an oblivious request, in addition to its content. It can also stand alone: because the content-addressed hash lets a user verify a package on arrival, concealing identity during retrieval is often sufficient by itself. The three mechanisms are independent layers of a shared privacy model, selected according to whether a user needs to conceal the content of a request, its origin, or both.

3.3 Builder Network

The peer transfer protocol describes how packages move. It does not describe how packages come to exist in the first place, or how peers verify that a received package is what it claims to be. Both issues are handled by the builder network.

3.3.1 Attestations

A *build attestation* is a signed claim that a given declarative specification, when evaluated and built, produces a specific output identified by its content hash. Any participant in the network may produce an attestation by performing the build locally and publishing the resulting hash along with their signature.

Attestations are published to a censorship-resistant network. Ethereum — or a dedicated layer-2 — is a natural candidate: attestations can be stored as transactions, where the resulting record may be independently verifiable, globally available, and irrevocable by any single party.

A user who receives a package from a peer can verify the package by checking its hash against published attestations. If multiple independent builders have attested to the same hash for the same specification, the user has reasonable confidence that the package is what it claims to be.

3.3.2 Builders

Confidence increases with the number of independent attestations. The system therefore benefits from a dedicated class of participants — *builders* — who perform builds professionally and publish attestations at scale. Builders may be motivated by direct economic incentive, by reputation, or by the desire to support specific software ecosystems they depend on.

A staking mechanism may be introduced to align builder incentives with network integrity. Builders post a bond when they publish an attestation, and the bond is forfeit if the attestation is successfully challenged. The structure of the bond, the duration of the staking period, and the conditions for slashing are matters of protocol design that remain open. What is fixed is the principle: a builder's claim must be backed by something the builder loses if the claim is false.

3.3.3 Challenges

Attestations must be challengeable. Without a challenge mechanism, a builder could publish false attestations with impunity, and the network's trust guarantees would collapse.

The challenge system proposed for Spaces is modeled on Cartesi's[22] Permissionless Refereed Tournament[23] (PRT) and its successor, Dave[24]. Any participant may bond a stake and submit a counter-claim asserting that a published attestation is incorrect. The original builder and the challenger then enter a structured verification game in which each commits to a computation hash — a Merkle tree of the entire build process[25] — and disputes are resolved by progressively narrowing the disagreement to a single computational step, which the blockchain can verify directly.

The PRT/Dave model has two properties that recommend it for the Spaces builder network:

1. It is *resistant to sybil attacks* — an adversary who spawns many false challengers gains no advantage, because the cost of each challenge grows with the number of challengers while the honest builder's defense grows only logarithmically.
2. It *permits anyone to participate* — the system does not depend on a privileged class of verifiers. Even users with modest computing resources can participate.

A builder whose attestation survives any number of challenges accumulates confidence over time. A builder whose attestation fails forfeits their bond. The protocol is designed to incentivize the continuous reduction of trust assumptions: as more challenges are mounted and more attestations are verified, the network's confidence in its package supply increases without any party being required to grant that confidence.

3.4 System Store

The package store, peer transfer protocol, builder network, and challenge system together describe how individual packages are distributed and verified. A space, however, is not a single package. It is a complete declarative specification, which references many packages and includes the configuration that binds them together. To make a space portable across machines, the specification itself must also be distributed.

The system store is the mechanism by which this occurs in a purely decentralized manner[26]. When a user has assembled a system of spaces, the complete configuration can be encrypted with a user's key and published to a censorship-resistant network. The encrypted artifact contains everything required to reconstitute the user's environment: the root space

22 https://docs.cartesi.io/get-started/cartesi-machine/
23 https://docs.cartesi.io/fraud-proofs/prt/prt-introduction/
24 https://github.com/cartesi/dave
25 The hashes of the Merkle tree are derived from the instruction tracing of the build process
26 Other system store backup solutions are permissible, such as simply using a friend's machine

configuration, the configurations of all sub-spaces, the references to packages and data containers, along with the access rules that govern them.

A user carrying only their private key can therefore reconstitute their environment on any machine running Spaces. The machine retrieves the encrypted configuration from the network, decrypts it with the user's key, resolves the package references against its local store or fetches them via the peer transfer protocol, and assembles the spaces according to the specification. The user's environment reappears around them. The hardware they happen to be using is incidental.

The system store also supports partial restoration. A user may publish individual spaces, subsets of their configuration, or differential updates rather than the entire system. This allows fine-grained recovery — a single sub-space restored without disturbing the rest of the environment — and supports use cases in which a user wishes to make a specific space available to others without exposing the totality of their system.

4 Local LLMs

Local LLM toolkits are the interface through which ordinary users assemble, modify, and operate a system that would otherwise require specialized expertise. This section describes how models are run on the user's own hardware, how their access is constrained, and the skills they progressively assume.

4.1 Local Harness

A *harness* is the framework that loads and runs a model on the user's machine. Local inference is now practical on consumer hardware. Open-source runtimes such as llama.cpp[27] perform inference across a wide range of processors and accelerators — from CPUs and Apple Silicon to consumer GPUs — and higher-level tools built on top of them manage model retrieval, quantization, and serving. A capable open-weight model can run on a single consumer GPU or an Apple Silicon laptop, and larger workloads can be offloaded to a dedicated inference machine within the user's own fleet.

In Spaces, a harness is declared like any other package. The model it loads, the resources it is granted, and the space it runs in are all specified in the configuration. A user may run one harness or many, each in a different space, each loaded with a different model.

4.1.1 Model Agnostic

The harness is agnostic to the model(s) it runs. Any compatible model can be loaded, and the choice is part of the user's specification rather than a property of the operating system. A user should be able to select models according to their task at hand from a library of models.

27 https://github.com/ggml-org/llama.cpp

Because the model is declared, it is also interchangeable. A user swaps one model for another by editing the configuration, in the same way they would swap any other component. Different spaces may run different models simultaneously. A research space may load a large, capable model for analysis, while a system-administration space runs a smaller model tuned for configuration tasks, and a communication space runs a model specialized for drafting text. No model is privileged. None is bound to the system. The user assembles the intelligence of their environment the same way they assemble everything else.

4.1.2 Security

It should not be assumed that models are trusted. They may behave in unexpected or adversarial ways. Security, therefore, does not depend on the behavior of the model. Rather, it depends on the boundaries of its space.

Models are subject to the same isolation assumptions enforced by the hypervisor. They have access only to their local data containers. A model tasked with drafting email, for example, only has access to the mail container and nothing else. In fact, it might not even have permission to send an email; only draft it. Alternatively, a model tasked with managing a space configuration would likely be restricted from all personal data altogether. An agent that traverses a trust boundary, passing a task from one space to another, does so only with the user's authorization.

4.2 Agent Command Interface (ACI)

The *agent command interface* (ACI) is how the user prompts their LLMs locally. The ACI is a successor to the command-line interface (CLI) and the graphical user interface (GUI). While the CLI requires the user to know the exact syntax of the commands they wish to run, and the GUI requires the designer to have anticipated and built a control for every action the user might take, the ACI requires neither. The user expresses intent in natural language — written or spoken — and the model translates that intent into operations on the system.

This inverts the conventional relationship between user and machine. Under the CLI and GUI, the user adapts to the affordances the system provides. Under the ACI, the system adapts to the intent the user expresses. It is available for any space or system as a declared package.

4.3 Skills

A *skill* is a capability the model exercises on the user's behalf, dispatched through the ACI and bounded by the permissions of the space in which the model runs.

4.3.1 System Administrator Assistance

The *system administrator* skill helps the user maintain their machine by abstracting technical operations with natural language. Such functions include but are not limited to:

- *Discovery and retrieval of software.* Rather than requiring the user to search the distributed package manager manually, the model crawls it, locating packages that satisfy a stated need, resolving their dependencies, and comparing available attestations to assess their trustworthiness. The user describes a capability they want; the model identifies the components that provide it and suggests it to the user.
- *Modification of the configuration.* When a system change is required, the model edits the relevant portion of the declarative specification and presents the change for approval. Because the configuration is declarative, the model operates on a coherent, inspectable artifact rather than issuing a sequence of imperative commands whose effects must be tracked. The change can be reviewed before it is applied and reversed after.
- *Diagnostics.* When the system misbehaves, the model inspects the configuration, identifies the likely cause, and proposes a remedy.

Throughout, the user remains in control. The model recommends and, upon approval, acts. Consequential operations require the user's confirmation.

4.3.2 Space Control

The *space control* skill directs the packages and services running within a space, and the files and data they hold. A user could, for example, prompt their ACI to locate a document and share it with a coworker. Anthropic's open-source Model Context Protocol[28] (MCP) offers a functional framework to enable space control. MCP defines an open client–server interface through which a model issues structured tool invocations against filesystems, services, and administrative functions.

4.3.3 Generative Tooling

The *generative* skill produces software rather than retrieving it. The user describes the capability they require, and the model generates the corresponding component. Here, the distinction between using a system and building a system further collapses.

Cumulative Generation as Applications

Generative tooling changes the nature of applications, broadly. Conventional platform applications are built for a general market, and their generality is a cost to the individual user. They come with features the user does not need, interactions designed for retention, and an interface that reflects the vendor's priorities rather than their own. A generated component is built for one user and one purpose. It does only what it was asked to do. Because it is purpose-built, it is more likely to be used and valued than a general-purpose application that must continually justify its own expansion.

28 https://modelcontextprotocol.io/docs/getting-started/intro

A generated application need not be monolithic. A single space can be filled with many small, individually generated, narrow components. Together, within the space that contains them, they constitute an application. The space, in the accumulation of all its micro-applications, becomes the portable artifact that a user shares and that another user remixes. What is exchanged is not a program but an environment, assembled from parts, each of which can be inspected, replaced, or regenerated.

Slint

Generated applications require generated interfaces. Slint[29], an open-source declarative GUI toolkit in the lineage of QML and XAML, is a natural candidate. Interfaces in Slint are described in a simple markup language rather than assembled through imperative construction. Generating a Slint interface is closer to generating a configuration than to writing a conventional program.

Slint is also well optimized for Spaces because its runtime is small and suitable for even modest hardware. It compiles to native code across desktop, mobile, embedded, and web targets from a single description, which suits environments that must reconstitute across heterogeneous machines. And because an interface is expressed as a declarative artifact rather than as machine-specific code, a Slint GUI is portable between users and machines in the same way the rest of a space is portable. An interface generated for one user can be shared with another, rendered on different hardware, and modified by the same model that produced it.

5 Implications

The operating system is the layer at which computing exerts its greatest cultural force. It determines what a machine is understood to be for, what its user is permitted to do, and who decides.

5.1 Be Your Own OS

The slogan "be your own bank" emerged in the early days of Bitcoin as shorthand for what cryptocurrency made possible. A private key, held by a user, granted full and unconditional control over digital assets — independent of any institution, jurisdiction, or counterparty. The slogan was not a description of a product. It was a description of a new relationship between a person and their wealth. Wealth had previously been custodied. Now it could be possessed.

We propose an analogous shift for operating systems. A private key, held by a user, grants full and unconditional control over a computational environment — independent of any hardware, vendor, or jurisdiction. Environments previously provisioned by third-parties and fixed to specific hardware can now be owned universally. We call this *being your own operating system.*

29 https://slint.dev/

5.2 Homelabs & Communal Hosting

Spaces do not need to be bound to a single machine. As an example, a user could easily run a space across three machines: a laptop on the go, a desktop workstation at home, and a NAS tucked in an office closet. Opening a document on the laptop reads it from the NAS. Prompting a model on the laptop runs it on the workstation's GPU. The user occupies one space, not three.

The network binding these machines can be declared alongside everything else. A few lines produce an encrypted mesh network[30]. Traffic between machines never traverses any third party's servers. No machine requires a public address, and the keys securing the mesh never leave the machines that generated them. The result is a private intranet.

The same composition extends across multiple users, not just multiple machines. Friends and family may contribute storage to one another's spaces, holding encrypted copies of data they cannot read. A household may share a single inference machine. A community may pool hardware for workloads no member could run alone. Access is granted cryptographically and revoked the same way. What platforms provide through custody, peers provide through composition.

Distributed, multi-device computing has been available to consumers only as a service. Google, Amazon, Apple, and Meta grant access to their machines through websites and applications, storing the data and running the compute on hardware they own. That is the product. Spaces let a user assemble the same architecture on machines they own instead.

5.3 Mass Deplatformization

Most people today cannot share a digital experience without a third party constructing it for them. Conversations require a messaging service. Shared photo albums require a storage provider. A community requires a forum, a server, or a network, each operated by a company whose participation is a condition of the community's existence. The provider is not a party to the relationship. It is the venue in which the relationship is permitted to occur.

The provider exists because users have lacked the means to assemble these capabilities themselves. The components that constitute most social applications — authentication, file storage, comment threads, image rendering, group messaging, access control — are not scarce. They exist as discrete, declarative modules, composable into a space on demand. With spaces, a user who wishes to share photos and comments with friends after a weekend trip does not depend on a third-party platform. They open a new space, generate the relevant components, and share access with their friends' public keys. The resulting environment is bespoke, governed entirely by the participants, and dissolved when it is no longer useful.

30 https://clan.lol/docs/

Bespoke social spaces compete directly with traditional platforms, like Facebook, X, and YouTube. A platform must serve enough users to justify the cost of building and operating it, which requires gathering heterogeneous communities into a shared context — the same feed, the same norms, the same interface. The resulting condition, which sociologists term *context collapse*, places audiences in a single digital room who would never have occupied one in physical life. A space carries no such requirement. It serves five people as well as five million, and there is no reason for the five to inhabit the environment built for the five million.

With spaces, a community's environment is configured by the community. Its interface reflects its norms. Its membership is its own. Its lifespan is whatever the participants decide, and its specification can be forked by anyone who wishes to begin again differently. The *platform* was a workaround for the absence of composable, ownable environments. With spaces, culture is no longer something a platform hosts. It is something a community builds.

5.4 Universal Files

A file is data separated from the means of interpreting it. A document, for example, only opens correctly if the recipient has the same application, in a compatible version, configured the same way. The file is portable. The environment that gives it meaning is not.

A space carries both. A document can be sent as a space containing the document, the application that renders it, and the configuration under which it was written. The recipient installs nothing. Opening the space assembles it, and the document appears exactly as its author intended.

Spaces also execute in isolation by default. A file received as a space runs inside a microVM, restricted to what the sender granted and the recipient permitted. Under the prevailing model, a downloaded file is opened by an application holding the full privileges of the user; one malformed document is enough to compromise the entire machine. Spaces are not only a more complete and universal means of sharing files. They are a safer one.

The file does not disappear. It arrives with its environment.

5.5 CROPS

The Ethereum Foundation has articulated a framework for evaluating sovereign digital infrastructure under the acronym CROPS[31]: *censorship-resistant, open-source (and free as in freedom), private, and secure*. CROPS was developed in the context of blockchain protocols, but the framework applies with equal force to computational environments. An environment that can be *censored* is not owned. An environment that is not *open* cannot be audited or modified

31 https://ethereum.foundation/ef-mandate.pdf#page=10

by its owner. An environment that is not *private* exposes its owner to surveillance and capture. An environment that is not *secure* can be compromised by adversaries.

Here it is critical to understand that we are not talking about an *operating system* that is CROPS. We are talking about *environments* that are CROPS. Operating systems are only the means to generate an environment. The environment, itself, is the artifact that each user personally owns, uses, and shares. It is their *spaces*.

5.5.1 Censorship Resistance

Censorship resistance is not merely the ability to run a program, like Tor or Bitcoin-Core, without restriction. It is the ability of the environment, itself — which is required to run them — to *survive*. An imperative environment exists only as the accumulated state of a machine, described nowhere, with no canonical form to restore from or compare against. Seizing the machine ends it permanently. Modifying it silently cannot be detected. A declarative specification backed up to a decentralized system store survives both attacks.

5.5.2 Open-Source and Free(dom)

The *freedoms* to study, modify, and redistribute a system presuppose there is an *artifact* to study, modify, and redistribute. While every open-source Linux distribution licenses its components freely, those that are imperative fail to produce any such artifact. The machine can be inspected — packages enumerated, files read — but no record states what the environment is. What an installer placed, what a build script did with root, what the owner altered by hand and forgot: none of it survives as anything but the machine's present condition.

By contrast a declarative specification is that artifact. It does not describe the environment. It is the environment, in a form that can be read, edited, and handed over to someone else, *freely and openly*.

5.5.3 Privacy

Privacy is not about isolation. It is about consent[32], which requires an environment that obeys its owner. A space bound to a private key cannot disclose beyond the scope of the declaration its owner signed. This is not a policy the environment observes, but the condition of its existence. An imperative environment can make no such claim, and not for want of good intentions. It simply has no owner in this sense. Its disclosures are the residue of whatever was last installed, configured, or updated, by whoever did so and whenever. There is no artifact against which the user could have consented in the first place.

32 https://xcancel.com/zooko/status/1068953367454539777

5.5.4 Secure

The objective of *security* is not to make more components trustworthy but to depend on fewer, which presupposes knowing which components those are. Imperative environments do not do this, while declarative ones do. A specification enumerates the trusted base exactly, in a form that can be reduced, until what remains is small enough to be a candidate for formal verification.

An enumerable surface also makes compromise portable. A user who is attacked need not surrender their machine to a researcher; they share the environment, which holds no data. It can be rebuilt anywhere, the attack reproduced, and the result attested against the hash of the component at fault — a proof that an environment is exploitable, published without disclosing what was taken from it. Every other user's exposure then reduces to a set-membership query: does my environment contain this component?

5.5.5 The Walkaway Test

The strongest expression of CROPS is a question. If every organization involved in building a system walked away, would the system continue to run? Would its users still be able to access, modify, and reproduce it? For nearly all software the answer is no. Maintainers stop maintaining. Registries go offline. Signing keys expire. The software rots, and then it stops.

Spaces are built to answer yes. A configuration is a specification rather than a service, and can be evaluated by anyone who holds it. Packages are distributed by peers and verified against attestations no party controls. The models that make the system usable run on the user's own hardware. The team that wrote this paper is a coordinating party like any other, and the architecture assumes we will eventually be gone — captured, defunded, or simply finished. Nothing in Spaces requires our continued participation.

6 Conclusion

Spaces defines how an environment is to be owned. Not just that a person may configure it, but that no one may take it. A space, once built, belongs to whoever holds the key — for as long as they hold it, on whatever hardware they can find, under whatever jurisdiction they happen to occupy. What they build in it is theirs. What they pass on is theirs.

Glossary

- **Agent-procured operating system** (ₐOS) — OS built by its user
- **Agent command interface** (ACI) — natural language interpreter to execute software
- **Build attestation** — signed claim a given declarative specification produces a hash
- **Builder network** — peers who build configurations and publish attestations at scale
- **CROPS** — property of being censorship resistant, open-source, private, and secure
- **Data container** — signed unit of data, attributed to owner's public key and addressable
- **Declarative configuration** — functional statement used to configure an environment
- **Harness** — framework that loads and runs a model on a user's machine
- **Hypervisor** — layer of software that creates and runs isolated virtual machines
- **Imperative configuration** — sequential commands used to configure an environment
- **Merkle tree** — repeatedly hashed pairs of a larger data set, defined by a parent hash
- **microVM** — lightweight virtual machine with its own kernel, isolated from the host
- **Model Context Protocol** (MCP) — defines an open client-server interface for LLMs
- **Nix** — declarative architectural language
- **NixOS** — declarative Linux distribution built from `nixpkgs` using Nix
- **nixpkgs** — monorepo package manager for Nix
- **Oblivious transfer** — cryptographic primitive to privately retrieve a file from a sender
- **Onion routing** — conceals traffic, routing requests through a circuit of relays
- **Platforms** — 3rd-party custodians of computational environments, grant user access
- **Provider-procured operating system** (ₚOS) — OS built by a third-party
- **Root space** — user's primary environment
- **Skill** (Local LLM) — capability the model exercises on the user's behalf
- **Slint** — declarative frontend language
- **Space** — ownable, computational environment
- **Store path** — derived from the cryptographic hash of its inputs
- **Sub-spaces** — isolated, purpose-specific environments that exist alongside the root
- **Sybil attack** — effort of single user to falsely represent themselves by multiple identities
- **System store** — copy of a complete configuration to assemble a system of spaces
- **Universal file** — data structure that declares its own execution environment

Acknowledgements

For their contributions, special thanks to Jonas Chevalier, Althea A, Kiran Lenk, and w.