Quuipu

QUUIPU RESEARCH & ARCHITECTURE SERIES

WHITEPAPER 02

Architecture

A Contextual Runtime for Adaptive Digital Experiences



Foundational Whitepaper · English

Quuipu Edition 2 · v1

Publication identity

Publication
Architecture

Series
Quuipu Research & Architecture Series

Publication number
Whitepaper 02

Edition
Quuipu Edition 2 · v1

Language
English

Historical edition
Cognisphere Edition 1

This publication succeeds Cognisphere Edition 1. Historical PDFs remain immutable.

Contents

01. Reference architecture

02. System context and boundaries

03. Quuipu Host

04. Quuipu Core

05. Context Engine

06. Memory and continuity

07. Semantic Layer

08. Cognitive Resolution

09. Experience Engine

10. State model

11. Canonical source

12. Provenance

13. Fallback and failure semantics

14. API surface

15. Events and observability

16. Multi-tenant architecture

17. Isolation and dependency architecture

18. Security, privacy and trust

19. Performance and runtime budgets

20. Integration pattern: existing site

21. Why it does not become another CMS

22. Reference deployment topologies

23. Engineering invariants and testability

24. Architecture status and roadmap

A. Reference contracts

B. Failure matrix

C. Implementation status matrix

D. Architecture invariants

Preamble

A Contextual Runtime for Adaptive Digital Experiences

> This publication succeeds Cognisphere Edition 1. Quuipu Edition 2 preserves the historical thesis while updating active brand and publication identity.

Preamble

Quuipu Architecture A Contextual Runtime for Adaptive Digital Experiences

Architectural Position

Inside the machine This paper describes Quuipu as an integrable contextual runtime. Its purpose is not to present a collection of features, but to define the responsibilities, boundaries, contracts, failure semantics and deployment patterns required for a site to become adaptive without losing its authority.

CMS / SOURCE SYSTEM = canonical authority EXPERIENCE = derived, bounded, reversible

Institutional Web ↓

Quuipu Host ↓

Experience Layer ↓

Cognitive Resolution ↓

Semantic Transformation ↓

Quuipu Core ↓

Context / Memory / State

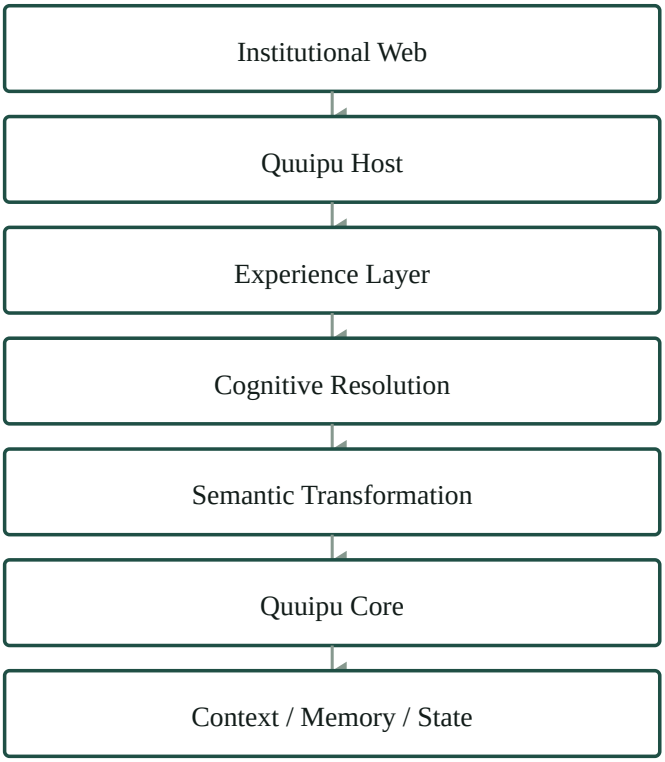


Figure 1 — Reference logical architecture.

Architecture At A Glance

Responsibility map Layer Owns Must not own

Host Integration, capabilities, security/content adapters Cognitive semantics

Experience Layer Projection, controls, visible state Canonical truth

Cognitive Resolution Intent-to-plan decision Editorial publishing

Semantic Layer Identity, relations, transformation fidelity Authentication

Core Orchestration, lifecycle, policy/fallback coordination CMS data model

Context Current bounded situation Permanent identity by default

Memory/State Continuity within explicit scope Unlimited hidden history

Architecture status legend Status Meaning

IMPLEMENTED Present in the existing Quuipu Institutional Web / Host line.

ARCHITECTURALLY DEFINED Consolidated contract or invariant used by the architecture.

REFERENCE / FUTURE Coherent target pattern; not represented as already implemented.

CHAPTER 01

Reference architecture

Quuipu Architecture describes a contextual runtime inserted between an existing digital host and the cognitive capabilities required to produce adaptive experiences. It starts from one rule: Quuipu does not own the site, does not own canonical content and does not replace the CMS.

The architectural chain The reference model is: Institutional Web → Quuipu Host → Experience Layer → Cognitive Resolution → Semantic Transformation → Quuipu Core → Context / Memory / State. The vertical direction expresses logical responsibilities rather than a monolith. The host controls what enters; the core coordinates what may happen; the experience layer makes the resolved state visible.

A runtime, not an editorial platform The runtime operates on contracts. It receives sources, capabilities, context and policy; it produces state and derived experiences. Authoring, editorial workflow, versioning and publishing remain in the site's infrastructure. This separation is what makes integration possible without replatforming.

Implemented, defined, future This V1 distinguishes three levels: IMPLEMENTED for properties already realized in the Quuipu Web/Host line; ARCHITECTURALLY DEFINED for consolidated contracts and principles; REFERENCE / FUTURE for coherent extensions not presented as already implemented. This distinction prevents the whitepaper from becoming imaginary documentation.

CHAPTER 02

System context and boundaries

Quuipu quality depends more on boundaries than on component count. A trustworthy cognitive runtime must know which data it may read, which capabilities it may invoke and which responsibilities belong to the host.

Host boundary The Quuipu Host is the integration boundary. It translates the site's specific world - routing, content, security context, lifecycle and capabilities - into public contracts consumed by the runtime. The core does not need to know CMS or host-framework details.

Security boundary The runtime receives a bounded security context, not raw credentials. Authentication and authorization remain host responsibilities; Quuipu can use only information and capabilities explicitly exposed.

Content boundary Sources enter through host content units and configuration. An authorized source retains identity, URI, version and metadata needed for provenance and canonicality. The runtime does not assume that everything technically reachable is semantically admissible.

Isolation Isolation prevents the runtime from depending on code, processes or datasets belonging to one host. The Quuipu/Discovery separation already embodies this rule: the host may depend on Quuipu contracts; Quuipu must not depend on the application embedding it.

Boundary Input Output Invariant

Content Authorized content units Canonical source refs No arbitrary repository access

Security Bounded security context Authorized capability decisions No raw credentials

Host Lifecycle + capabilities Runtime-compatible contracts Core stays host-agnostic

Tenant Tenant-scoped config/state Isolated execution context No cross-tenant reads

CHAPTER 03

Quuipu Host

The Host is the architectural adapter that lets Quuipu enter different sites without imposing one framework, CMS or deployment model.

Lifecycle The reference lifecycle follows create → configure → initialize → start → ready → shutdown → dispose. Separating these phases distinguishes construction, configuration, dependency activation, operational availability and resource release.

Host capabilities Capabilities declare what the site allows the runtime to do: read a source, resolve a link, obtain context, activate a transformation or access an authorized function. A missing capability must not be simulated.

Browser host In the Institutional Web, consumption flows through a facade and web-host adapter toward the browser-host/distribution boundary. This keeps the public site separate from internal runtime implementation and enables progressive enhancement.

No framework capture The Host prevents React, Astro, Vite, a CMS or an authentication system from becoming constitutional dependencies of the core. Technologies may change; the cognitive contract should remain stable.

Host Adapter Runtime Experience

create construct contracts created baseline UI

configure provide capabilities configured canonical content

initialize/start bind dependencies initializing progressive activation

ready expose readiness ready cognitive controls enabled

shutdown/dispose release bindings closed canonical UI remains

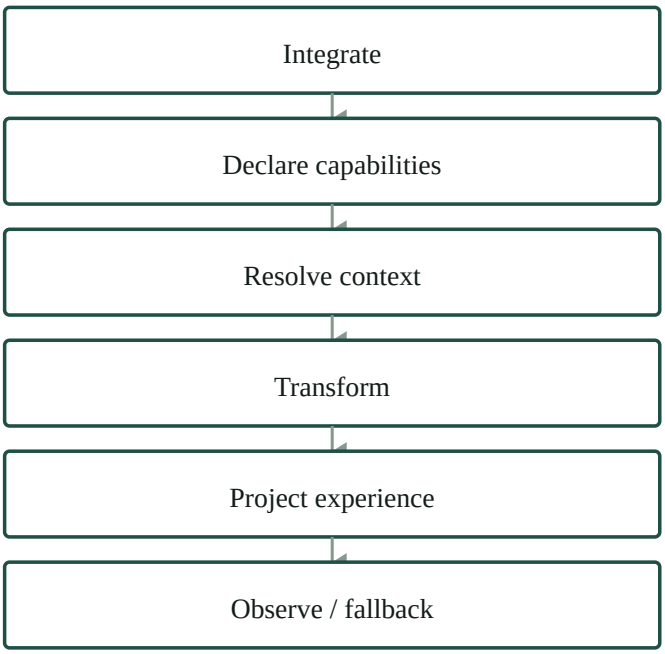


Figure 2 — Lifecycle integration sequence.

CHAPTER 04

Quuipu Core

The Core is the runtime coordination authority. It is neither an AI model nor a database: it governs the transition among authorized input, cognitive state, resolution, transformation and experience emission.

Responsibilities The Core coordinates lifecycle, capabilities, policy, state, resolution and fallback. It keeps 'may I perform this operation?' separate from 'is this transformation semantically appropriate?' and from 'how should this appear in the interface?'.

Orchestration over ownership The Core orchestrates specialized components without owning their sources. The Context Engine governs context, the Semantic Layer governs identity and relationships, and the Experience Engine governs visible form. This division reduces coupling and makes responsibilities testable.

Deterministic shell, probabilistic capabilities The runtime may contain probabilistic capabilities, but its control shell remains deterministic: timeout, capability state, provenance binding, error classification and fallback cannot depend on model creativity.

CHAPTER 05

Context Engine

The Context Engine builds the minimum representation of the current situation required to resolve an experience.

Context sources Context may include declared intent, current page or unit, language, mode, available capabilities, session state and host constraints. Every field should have an operational reason.

Context assembly Assembly combines explicit signals and observable state into a Context Snapshot. A snapshot is preferable to indiscriminate global-state access because it makes resolution input reproducible and inspectable.

Context validity The engine distinguishes missing, partial, stale and conflicting context. Incomplete context can trigger clarification or a conservative experience; it should not automatically be filled by invisible inference.

Context minimization The principle is to collect what the next step needs, not to build a total profile. This also enables no-telemetry/no-persistence experiences when memory beyond the session is unnecessary.

CHAPTER 06

Memory and continuity

Memory is separate from current context. Quuipu can preserve continuity without assuming that every interaction must be persisted.

Ephemeral state Ephemeral state holds what the interaction needs: current intent, mode, active source, traversed steps, request state and temporary provenance.

Session memory Session memory can preserve continuity across pages or transformations. It needs explicit scope, lifetime and reset.

Persistent memory Persistent memory is an optional and more sensitive capability. This V1 treats it as a governed extension rather than a product prerequisite. Persistence, tenant scope, retention and consent belong to deployment policy.

Continuity contract Continuity should be reconstructable from declared state rather than hidden side effects. The runtime should know which intent continues, which source was used and which transformation is active.

CHAPTER 07

Semantic Layer

The Semantic Layer protects the relationship among canonical content, concepts and derived representations.

Semantic identity Entities, concepts and relationships retain identity across modes. Original, Simple and Step-by-step remain projections of one substrate rather than independent content.

Canonical relationship map Explore can use a canonical relationship map for guided navigation. The map must not invent links merely because they are semantically plausible: admissible relationships are bounded by sources and host governance.

Transformation contract A semantic transformation receives source, representation goal, context and policy; it produces derived output plus provenance and state. The contract must represent partial success, refusal and fallback.

Semantic fidelity Simplifying, reordering or decomposing does not authorize removing conditions that change meaning. Fidelity is therefore a system requirement rather than a stylistic preference.

CHAPTER 08

Cognitive Resolution

Cognitive Resolution is the decision core that turns contextualized intent into an executable experience plan.

Resolution pipeline The reference pipeline is: Context Assembly → Intent Resolution → Source Eligibility → Semantic Resolution → Candidate Set → Policy Evaluation → Experience Plan → Provenance Binding → Output Contract.

Intent resolution Intent is normalized without erasing the user's wording. Ambiguity may yield multiple candidates or clarification; out-of-scope intent should be declared as such.

Source eligibility Before using a source, the runtime checks authorization, relevance, availability and policy compatibility. This separates retrieval from authority.

Experience plan The resolution result is not yet UI. It is a plan: mode, sources, permitted transformations, sequence, confidence/status, timeout budget and fallback target.

Context Assembly ↓

Intent Resolution ↓

Source Eligibility ↓

Semantic Resolution ↓

Candidate Set ↓

Policy Evaluation ↓

Experience Plan ↓

Provenance Binding ↓

Output Contract

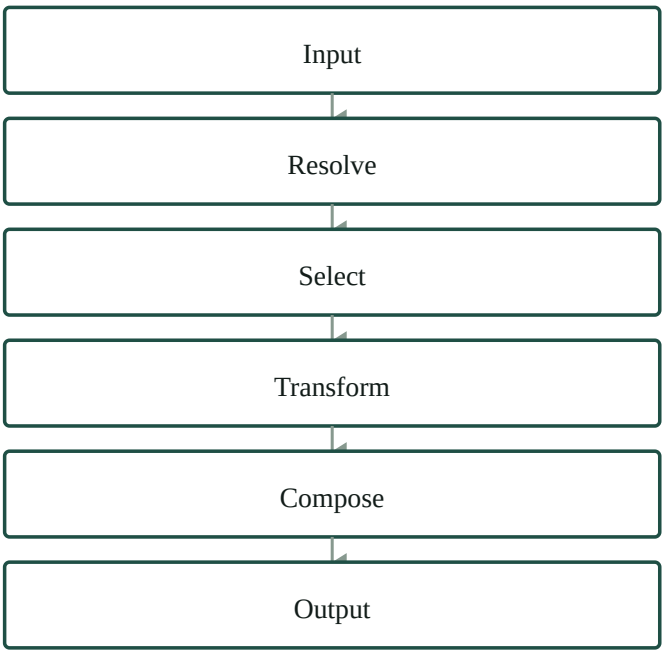


Figure 3 — Cognitive Resolution reference pipeline.

CHAPTER 09

Experience Engine

The Experience Engine translates the cognitive plan into visible experience without hiding runtime state.

Experience Experience lets a person declare what they are trying to understand, activates the pipeline and makes the Standard → Quuipu transformation visible.

Understand Understand exposes Original, Simple and Step-by-step representations. Controls remain accessible, source remains available and noscript fallback preserves canonical content.

Explore Explore uses paths and topics bounded by canonical relationships. It is not open chat: the person chooses where to go and the runtime maintains continuity.

Progressive enhancement Canonical UI renders first. The runtime activates afterwards; if unavailable, the page still works. This principle is already part of the Institutional Web line.

CHAPTER 10

State model

The state model makes the runtime observable and prevents the interface from confusing processing, success and fallback.

Capability state Relevant capabilities can be unavailable, initializing, ready, active, degraded or failed. Exact taxonomy may evolve; the principle is explicit consumable state.

Request state A cognitive request can move through idle, resolving, transforming, composing, complete or fallback. These states feed aria-live, disabled controls and diagnostics.

No hidden success A UI must not label an experience Quuipu-enhanced when the runtime was not actually invoked. The Web line already requires invocation proof and capability state to distinguish real activation from static content.

State ownership State belongs to the layer that produces it. The Experience Engine projects state rather than inventing it. The Core aggregates without turning every component into one global store.

State Meaning Typical UI projection

idle No active cognitive request Standard experience

resolving Context/intent/source resolution Busy + aria-live

transforming Derived representation in progress Mode pending

complete Valid result bound to provenance Enhanced experience

degraded Some capability unavailable Conservative experience

fallback Cognitive path abandoned Canonical content

failed Non-recoverable request failure Explicit error + source path

CHAPTER 11

Canonical source

The canonical source is the editorial authority from which experience derives. It is the anchor that enables adaptation without loss of trust.

Identity Each canonical unit should be stably identifiable within the host through URI, content ID, version or equivalent. The concrete choice depends on infrastructure.

Canonical-first rendering Canonical content remains available before or independently of cognitive transformation. On a public site this protects SEO, accessibility, deep linking and resilience.

Derived representations A derived representation does not replace the canonical record. It is a runtime artifact bound to a source and context. Its validity can expire when the source changes.

Staleness The runtime should detect or at least represent staleness risk. Cache and memory must not make an authoritative source update invisible.

CHAPTER 12

Provenance

Provenance connects every cognitive output to the sources and activities that produced it.

Provenance record A minimal record can include source identity, source version/hash when available, transformation type, timestamp/session scope, capability/model identifier and status. Not every field belongs in the UI, but it should exist at the appropriate layer.

User-facing provenance The interface should expose at least the derived nature of output and a path to source. High-trust contexts may also expose date/version or transformation details.

Operational provenance For debugging and audit, provenance reconstructs which source and pipeline produced the output. This differs from behavioral telemetry: derivation can be traced without profiling the user.

Provenance binding Binding occurs before final emission. Output that cannot be sufficiently connected to origin should be degraded, qualified or refused.

CHAPTER 13

Fallback and failure semantics

Fallback is not an edge case: it is a constitutional property of a cognitive layer placed over a site that already works.

Timeout The Web line uses bounded timeout for cognitive activation. The exact value is configurable; the principle is that users are not trapped waiting for an optional capability.

Source failure If a source is unavailable or ineligible, the runtime does not compensate with an unauthorized source. Fallback can be already-rendered canonical content or an explicit inability state.

Transformation failure If a transformation fails, Original remains available. Failure of Simple should not compromise Explore or the host page when capabilities are independent.

Degraded mode A runtime may be partially operational. Exposing degraded state lets the Experience Engine choose a conservative path rather than turning every problem into a global error.

Failure Do not Preferred behavior

Timeout Block indefinitely Return canonical/fallback

Missing source Invent a substitute Declare source unavailable

Semantic uncertainty Present as authoritative Qualify/refuse/return Original

Capability failure Fail whole site Degrade only dependent mode

Provenance failure Hide derivation gap Do not emit as trusted enhancement

CHAPTER 14

API surface

Quuipu APIs should be smaller than the internal implementation. The public contract protects hosts and partners from core evolution.

Runtime creation The reference public contract includes runtime creation and HostCapabilities provision. Construction should not automatically trigger irreversible effects.

Commands and queries APIs can separate commands - activate, resolve, transform, shutdown - from queries - getState, getCapabilities, getProvenance. This makes side effects and reads explicit.

Typed contracts Context, content unit, resolution request, experience plan, result, error and provenance should be versioned types. Contract compatibility matters more than internal module stability.

API versioning Evolution should favor additive change and capability negotiation. A new feature should not force an older host to immediately implement new methods.

Contract Direction Purpose

createRuntime Host → Core Construct runtime

configure Host → Core Provide bounded config/capabilities

resolve Experience → Core Request cognitive plan

transform Core → Semantic capability Produce derived representation

getState Experience → Core Read operational state

getProvenance Experience/Audit → Core Inspect derivation

shutdown/dispose Host → Core Terminate lifecycle safely

CHAPTER 15

Events and observability

An integrated runtime needs to communicate change without coupling the host to core details.

Domain events Events such as runtime.ready, capability.changed, resolution.started, resolution.completed, transformation.failed or fallback.entered can form an observable surface. These names are reference design and require versioning in implementation.

UI events versus domain events A click is not necessarily a domain event. The Experience Layer translates interactions into commands; the Core emits events about operational meaning.

Observability without surveillance Health, latency, errors and fallback can be measured without recording sensitive intent content. Technical telemetry and behavioral analytics remain separate.

Correlation A resolution can use an ephemeral correlation ID to connect logs, provenance and events. It should not automatically become a persistent user identifier.

CHAPTER 16

Multi-tenant architecture

For organizational and platform adoption, Quuipu must be able to separate configuration, sources, policy and state among tenants.

Tenant boundary A tenant is an administrative isolation domain. Content sources, policy, capabilities and memory scope of one tenant must not be readable by another.

Configuration layering Configuration can combine runtime defaults, deployment policy, tenant configuration and session context. A more specific layer cannot violate constitutional invariants.

Data plane and control plane As a reference architecture, separating a control plane for configuration, versions and policy from a data plane executing resolution and transformation is useful. This is proposed for enterprise deployment and is not presented as already implemented.

Tenant-aware provenance Provenance, cache and memory include tenant scope. A cache key that ignores tenant identity can break isolation even when content is public.

Control Plane Tenant Boundary Data Plane Host

policy/version tenant A resolution/transform site A

policy/version tenant B resolution/transform site B

shared invariants isolated config/state isolated execution independent canonical source

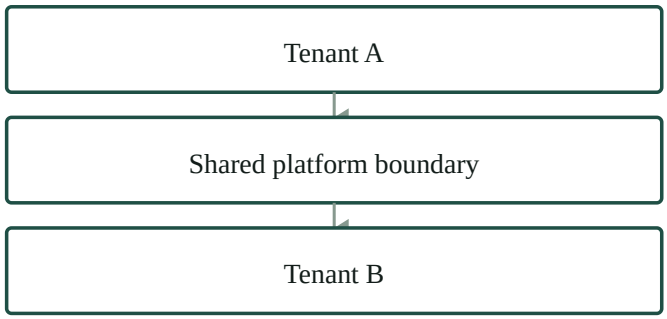


Figure 4 — Reference multi-tenant separation.

CHAPTER 17

Isolation and dependency architecture

Isolation is what makes Quuipu an integrable product rather than a feature embedded in one site.

Dependency direction The allowed direction is Host/Application → Quuipu public contracts. The Core does not import host components. Adapters live at the boundary and can be replaced.

Package boundaries Runtime bootstrap, contracts, browser-host distribution and institutional web can evolve as separate packages. Site builds depend on required public artifacts without dragging the whole repository or development tooling.

No Discovery dependency Separation from Discovery is an important architectural precedent: Quuipu was extracted to eliminate static dependencies and treat the host as consumer rather than owner of the runtime.

Test boundaries Each boundary should have contract tests: lifecycle, capability negotiation, content/security adapters, fallback and serialization. End-to-end tests then verify that real experience respects the contract.

CHAPTER 18

Security, privacy and trust

Contextual adaptation increases system responsibility because it introduces state and transformations over authoritative content.

Least capability The runtime receives only required capabilities. It should not gain general browser, CMS or credential access when a specific adapter is enough.

Data minimization Intent and context can contain sensitive information even when they are not treated as a profile. Deployments minimize retention, logging and propagation.

Policy enforcement Source, transformation, tenant and security policy is enforced before execution, not merely documented. An AI model cannot bypass policy because it produced plausible output.

Trust surface Users can see when Quuipu is active, which mode is in use, when fallback occurred and how to reach the canonical source.

CHAPTER 19

Performance and runtime budgets

A cognitive layer must respect that it is a guest of the experience rather than its sole owner.

Startup budget Activation should not block canonical rendering. Lazy activation and progressive enhancement reduce initial cost.

Resolution budget Each request has a time budget. When cost exceeds the threshold, the runtime can return a partial result or fallback rather than degrading the experience indefinitely.

Caching Semantic or transformation caches can reduce latency but are invalidated against source version, tenant, policy and transformation version.

Bundle discipline In the Web host, every capability adds bundle cost. The Institutional Web line already treats bundle delta as a release metric, reinforcing that cognition must not erase standard Web qualities.

CHAPTER 20

Integration pattern: existing site

The primary adoption pattern is incremental: Quuipu is added above an existing site, not in place of it.

Step 1 - establish canonical surface Identify authoritative sources, journeys and capabilities. No transformation should precede a stable baseline.

Step 2 - add Host adapter Map content, routing, security and lifecycle into Quuipu contracts. The adapter is the only point that understands both architectures.

Step 3 - activate one experience Begin in a high-friction area with one bounded mode. This limits risk and validates provenance, fallback and accessibility.

Step 4 - expand capabilities Add Context Resolution, Explore, memory or other capabilities only after stabilizing the previous contract.

Step 5 - operate and govern Monitor errors, latency, fallback and fidelity; version policy and contracts; keep the canonical path as permanent control.

Canonical site baseline ↓

Host adapter ↓

One bounded cognitive mode ↓

Resolution + provenance ↓

Fallback validation ↓

Capability expansion ↓

Operational governance

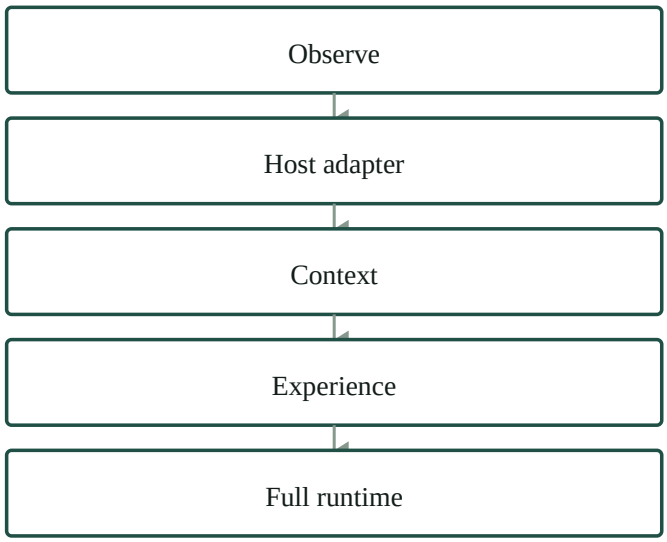


Figure 5 — Incremental adoption path.

CHAPTER 21

Why it does not become another CMS

The decisive CTO question is whether Quuipu introduces a second editorial platform. The architecture is designed precisely to avoid that outcome.

CMS owns truth The CMS or source system continues to own content, workflow, revisions, editorial permissions and publishing.

Quuipu owns runtime interpretation Quuipu owns temporary context resolution and experience composition. Derived outputs do not automatically become editorial records.

No content fork If a Simple representation should become permanent content, it must re-enter the host's editorial workflow. The runtime does not silently create a second truth.

Replaceable layer A site must be able to disable Quuipu and continue to exist. This is the simplest test that the cognitive layer has not captured the architecture.

Concern CMS / source system Quuipu

Authoring Owns Does not own

Editorial approval Owns Does not own

Canonical version Owns References

Context resolution Does not own Owns at runtime

Derived representation May publish if approved Produces temporarily

Navigation adaptation Static/configured Can compose contextually

Fallback Canonical experience Returns to canonical experience

CHAPTER 22

Reference deployment topologies

Quuipu can be deployed in several topologies as long as boundaries remain invariant. This section is reference architecture, not a claim that every topology is already available.

In-process / browser-enhanced Part of the runtime lives close to the Web experience and uses browser adapters. It suits bounded capabilities and progressive enhancement.

Service-backed runtime Resolution or transformation can run in remote services while only the host adapter and projection layer remain in the browser. This requires network policy, auth and latency budgets.

Hybrid Context and UI state can remain client-side while expensive transformations are delegated. Provenance crosses both sides.

Enterprise multi-tenant A shared control plane can manage policy and versions while isolated data planes execute tenant requests. This is a future pattern consistent with the isolation architecture.

CHAPTER 23

Engineering invariants and testability

Architecture must be testable, not merely describable.

Lifecycle invariants create is not ready; shutdown precedes dispose; a capability cannot be invoked before availability. These constraints make integration predictable.

Canonical invariants Every derived output has a source or explicitly declares the absence of an authorized derivation. Original remains reachable.

Fallback invariants A cognitive failure does not make the host unusable. Fallback does not loop and is not presented as success.

Isolation invariants No tenant crosses another tenant boundary; no adapter exposes raw credentials; no core package depends on host code.

CHAPTER 24

Architecture status and roadmap

Credible documentation distinguishes what exists from what the architecture enables.

Implemented baseline The Quuipu Institutional Web line has implemented app bootstrap, canonical content, experience logic, Web Host integration, Cognitive Experience Activation, fallback, status UI and progressive enhancement; browser-host boundaries and host contracts are part of the technical base.

Architecturally defined Canonical-first, provenance, context/state separation, semantic transformation, lifecycle, capability boundaries and isolation form the consolidated architectural model used in this paper.

Reference / future Multi-tenant control plane, enterprise persistent memory, distributed event fabric, policy service and some service-backed topologies are described as reference evolutions requiring dedicated implementation and certification.

The architectural promise The value is not having the most components. It is being able to add cognitive capabilities to a digital experience while keeping source, host, security and control separate.

CHAPTER A

Reference contracts

The following pseudo-contracts are architectural notation, not a promise of exact production API names. HostCapabilities content: ContentCapability security: SecurityCapability navigation?: NavigationCapability context?: ContextCapability clock?: ClockCapability ResolutionRequest intent: DeclaredIntent context: ContextSnapshot sourceScope: SourceRef[] requestedMode: ExperienceMode policy: PolicyRef ExperiencePlan status: ResolutionStatus sources: EligibleSource[] steps: ExperienceStep[] fallback: FallbackTarget provenanceSeed: ProvenanceSeed DerivedExperience representation: ExperienceRepresentation state: RuntimeState provenance: ProvenanceRecord canonicalReturn: SourceRef

CHAPTER B

Failure matrix

Condition Core decision Experience projection Canonical guarantee

No context Resolve conservatively / request Standard or guided prompt Source remains available clarification

Ambiguous intent Return alternatives Choice UI No silent assumption

Source stale Invalidate/qualify Refresh or warning Current canonical wins

Transformation timeout Enter fallback Original Host remains usable

AI unavailable Use deterministic path if allowed Reduced capability No fake AI result

Policy refusal Refuse transformation Explain bounded limitation Canonical source

Tenant mismatch Reject No content projection Isolation preserved

Provenance incomplete Do not trust-enhance Fallback/qualified result Source path retained

CHAPTER C

Implementation status matrix

This matrix deliberately separates the existing Quuipu baseline from reference enterprise extensions.

Area Status Whitepaper treatment

Institutional Web bootstrap/content IMPLEMENTED Existing baseline

Web Host integration / lifecycle IMPLEMENTED Existing baseline

Cognitive Experience Activation IMPLEMENTED Existing baseline

Canonical-first + fallback IMPLEMENTED / DEFINED Existing behavior + invariant

Experience / Understand / Explore DEFINED / evolving product surface Architecture contract

Context / state separation ARCHITECTURALLY DEFINED Core model

Provenance model ARCHITECTURALLY DEFINED Trust contract

Persistent memory REFERENCE / FUTURE Optional governed capability

Enterprise control plane REFERENCE / FUTURE Deployment pattern

Distributed event fabric REFERENCE / FUTURE Operational extension

Multi-tenant data plane REFERENCE / FUTURE Isolation pattern

CHAPTER D

Architecture invariants

A1 · Host sovereignty The host owns canonical content, authentication and publishing.

A2 · Core independence Core packages do not depend on host application code.

A3 · Canonical-first Derived experience never removes the canonical path.

A4 · Bounded capability Unavailable capability is not simulated.

A5 · Explicit state Processing, degraded, fallback and failure are observable.

A6 · Provenance before trust Trusted derived output is bound to source.

A7 · Failure containment Cognitive failure does not become host failure.

A8 · Tenant isolation Configuration, state and data remain tenant-scoped.

A9 · Memory optionality Persistent memory is not required for contextual adaptation.

A10 · Replaceability Quuipu can be removed without destroying the host's canonical function.

Architecture A contextual runtime that adds cognition without taking ownership of the Web.

HOST SOVEREIGNTY · BOUNDED INTELLIGENCE · CANONICAL-FIRST · PROVENANCE · FALLBACK