

ORANGE

Describe it. Orange compiles it.

A Local-First, Browser-Native Architecture for Schema-Constrained AI Workflow Automation

EXECUTIVE SUMMARY

ORANGE is a 100% local-first AI automation studio that runs entirely inside a web browser. A user describes a workflow in natural language. ORANGE compiles that description into a typed, validated directed acyclic graph (DAG) and executes it through a deterministic runtime. No backend, no API keys, and no per-token billing are involved at any stage.

The system rests on three architectural commitments:

- **On-device inference.** A quantized `Qwen2.5-0.5B-Instruct` model runs through WebGPU and WebLLM, inside an isolated Web Worker.
- **Compilation, not code generation.** The model emits a constrained JSON graph, which a deterministic validator checks and repairs. The model never writes executable code.
- **Closed-world execution.** The runtime interprets a fixed set of five node types and nothing else. Network access is limited to an allowlist of keyless public APIs.

The result is an automation platform with zero server cost, a minimal attack surface, and shareable workflows that need no database.

Abstract

Modern AI automation tools depend on centralized cloud inference. This dependency creates recurring per-token costs, API key management overhead, data-residency exposure, and a single point of failure for every workflow. It also invites a dangerous pattern: asking a model to generate arbitrary code and executing it. ORANGE removes these constraints with a local-first architecture. A small language model runs on the user's GPU through WebGPU, and its output is forced into a typed JSON DAG. A deterministic validator and self-repair loop guarantee structural correctness, and an interpreter executes only a closed set of node types against an allowlisted set of APIs. Workflows serialize into the URL hash itself, so sharing requires no backend. The outcome is an automation studio that is private by construction, free to operate, and safe by design.

1. Introduction & The Local-First Paradigm

1.1 The Problem with Cloud-Coupled Automation

Conventional AI workflow builders are thin clients over remote inference endpoints. That coupling carries structural costs:

Concern	Cloud-Coupled Systems	ORANGE
Cost model	Per-token or per-seat billing	\$0, fixed
Credentials	API keys stored, rotated, leaked	None required
Privacy	Prompts transit third-party servers	Prompts never leave the device
Availability	Dependent on vendor uptime and quotas	Works offline after first model load
Operational burden	Servers, scaling, observability	Static file hosting

1.2 WebGPU: Compute as a Browser Primitive

WebGPU is the modern browser API for low-level GPU access. Unlike WebGL, which was designed for rasterization, WebGPU exposes compute shaders, explicit buffer management, and a command-queue model close to Vulkan, Metal, and Direct3D 12. This makes in-browser neural network inference practical, because matrix multiplications and attention kernels run as native GPU workloads instead of being emulated through graphics pipelines.

1.3 WebLLM and Model Selection

WebLLM is an in-browser LLM inference runtime that compiles model kernels to WebGPU. ORANGE uses it to load [Qwen2.5-0.5B-Instruct](#), a roughly half-billion-parameter instruction-tuned model. The selection criteria are deliberate:

- **Small footprint.** The quantized weights are small enough to download once and cache in browser storage, so subsequent sessions start without network transfer.
- **Sufficient capability for a constrained task.** ORANGE does not ask the model to reason open-endedly. It asks the model to map intent onto a narrow, typed schema, a task where small models perform well when paired with validation.

- **Broad hardware reach.** A 0.5B model fits within the memory budgets of integrated GPUs, widening the addressable device population.

The architectural principle is that **intelligence is cheap when the problem is constrained**. ORANGE trades model size for strong guarantees at the system level.

1.4 The \$0 Architecture

ORANGE consists of static assets: HTML, CSS, and vanilla JavaScript. It uses no framework runtime, no build-time dependency tree, and no server process. Consequently:

- **Hosting** is any static file host or CDN.
- **Inference cost** is borne by hardware the user already owns.
- **Scaling** is inherent: each additional user brings their own compute.
- **Secrets management** is eliminated, because there are no secrets.

1.5 The Living Orb

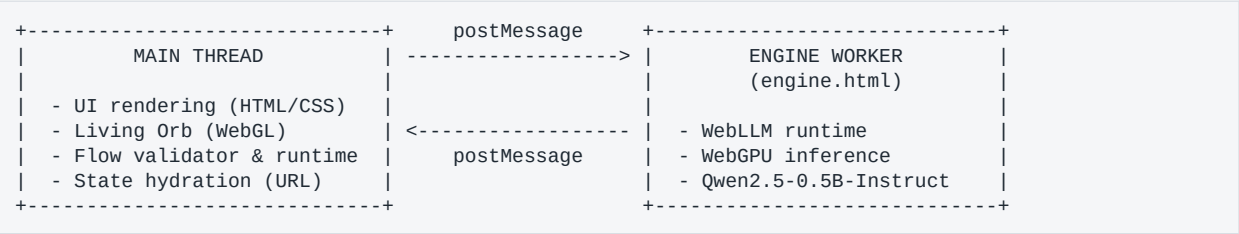
The interface centerpiece is the "Living Orb," a dynamic UI component rendered with custom WebGL fragment shaders. The orb reflects system state (idle, compiling, executing) as a continuously animated visual. Because rendering is GPU-driven and inference is isolated in a worker (Section 2), the orb maintains a steady 60fps even while the model is saturating the GPU queue for inference.

2. The Web Worker Architecture

2.1 Design Rationale

LLM inference is long-running and computationally intensive. Executing it on the main thread would block event handling, layout, and paint, producing a frozen interface precisely when users expect feedback. ORANGE isolates all heavy AI compute in a dedicated Web Worker context, `engine.html`, and communicates with it exclusively through message passing.

2.2 Thread Topology



2.3 Message Protocol

The boundary between threads is a narrow, explicit message contract. A representative shape:

```
// Main thread -> Engine: request compilation
engine.postMessage({
  type: 'compile',
  id: requestId,
  prompt: userDescription
});

// Engine -> Main thread: progress and result events
// { type: 'load-progress', progress: 0.62 }
// { type: 'token', id, delta }
// { type: 'result', id, json }
// { type: 'error', id, message }
```

Key properties of this design:

- **Non-blocking by construction.** The main thread only sends requests and handles events. It never awaits inference synchronously.
- **Streamed feedback.** Model load progress and generated output stream back as discrete messages, driving progress indicators and the Orb's visual state.
- **Fault isolation.** A fault in the inference context cannot corrupt UI state. The main thread can surface the error and recover.
- **Trust boundary.** The worker returns text. The main thread treats that text as *untrusted input* and routes it through the validator before anything else happens.

2.4 Frame-Budget Discipline

At 60fps, each frame has a budget of approximately 16.7ms. By confining inference to a separate context and keeping the main thread limited to rendering, validation, and DAG interpretation, ORANGE preserves this budget. Shader animation, input handling, and result rendering remain fluid regardless of model activity.

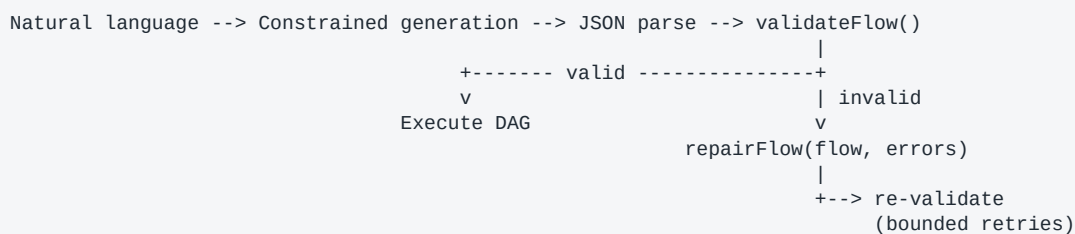
3. Schema-Constrained Decoding: The Compiler

3.1 From Language to Graph

ORANGE treats natural language as *source code* and the model as a *compiler front-end*. The target representation is a **Flow**: a typed JSON DAG whose vertices are nodes and whose edges express data dependency. The model is forced to emit this structure, not free-form prose or executable code. An illustrative Flow:

```
{
  "name": "Weather Brief",
  "nodes": [
    { "id": "n1", "type": "trigger", "params": {} },
    { "id": "n2", "type": "http-get",
      "params": { "url": "https://api.open-meteo.com/v1/forecast?...&current_weather=true" } },
    { "id": "n3", "type": "transform", "params": { "path": "current_weather.temperature" } },
    { "id": "n4", "type": "output", "params": { "label": "Temperature (C)" } }
  ],
  "edges": [
    { "from": "n1", "to": "n2" },
    { "from": "n2", "to": "n3" },
    { "from": "n3", "to": "n4" }
  ]
}
```

3.2 The Compilation Pipeline



3.3 Why Validation Is Non-Negotiable

A 0.5B-parameter model will occasionally hallucinate: inventing node types, omitting required parameters, referencing nonexistent node IDs, or emitting malformed JSON. ORANGE does not treat this as a failure of the model but as an *expected input condition* for the compiler. Correctness is therefore enforced by a deterministic layer rather than hoped for from a probabilistic one.

3.4 The Validator

`validateFlow(flow)` is a pure function. It accepts a candidate Flow and returns a list of structured errors, empty if the Flow is valid. It enforces:

- **Closed type set.** Every node `type` must belong to `{trigger, http-get, transform, filter, output}`.
- **Required parameters.** Each node type declares mandatory params and their types.
- **Referential integrity.** Every edge must reference existing node IDs, and node IDs must be unique.
- **Acyclicity.** The graph must contain no cycles, so execution is guaranteed to terminate.
- **Structural rules.** For example, a Flow requires an entry `trigger` and at least one `output`.

```

const NODE_TYPES = new Set(['trigger', 'http-get', 'transform', 'filter', 'output']);

function validateFlow(flow) {
  const errors = [];

  if (!flow || !Array.isArray(flow.nodes) || !Array.isArray(flow.edges)) {
    return [{ code: 'SHAPE', message: 'Flow must contain nodes[] and edges[]' }];
  }

  const ids = new Set();
  for (const node of flow.nodes) {
    if (ids.has(node.id)) {
      errors.push({ code: 'DUP_ID', nodeId: node.id,
        message: `Duplicate node id "${node.id}"` });
    }
    ids.add(node.id);

    if (!NODE_TYPES.has(node.type)) {
      errors.push({ code: 'BAD_TYPE', nodeId: node.id,
        message: `Unknown node type "${node.type}"` });
      continue;
    }
    errors.push(...validateParams(node));
  }

  for (const edge of flow.edges) {
    if (!ids.has(edge.from) || !ids.has(edge.to)) {
      errors.push({ code: 'BAD_EDGE', edge,
        message: 'Edge references a nonexistent node' });
    }
  }

  if (hasCycle(flow)) {
    errors.push({ code: 'CYCLE', message: 'Flow must be acyclic' });
  }

  return errors;
}

```

3.5 The Self-Repair Loop

When validation fails, ORANGE does not discard the result. `repairFlow(flow, errors)` applies deterministic corrections where the fix is unambiguous, and where it is not, it feeds the structured error list back to the model as targeted feedback for another constrained attempt.

```

const MAX_REPAIR_ATTEMPTS = 3;

function repairFlow(flow, errors) {
  const repaired = structuredClone(flow);

  for (const err of errors) {
    switch (err.code) {
      case 'BAD_EDGE':
        // Drop edges that point to nonexistent nodes
        repaired.edges = repaired.edges.filter(
          e => !(e.from === err.edge.from && e.to === err.edge.to)
        );
        break;
      case 'DUP_ID':
        // Re-key duplicate nodes deterministically
        reassignDuplicateId(repaired, err.nodeId);
        break;
      case 'BAD_TYPE':
        // Unknown types cannot be guessed safely; flag for model re-prompt
        repaired.__needsRegeneration = true;
        break;
      // ...additional rules per error code
    }
  }
  return repaired;
}

async function compile(prompt) {
  let flow = await generateFlow(prompt);
  for (let attempt = 0; attempt < MAX_REPAIR_ATTEMPTS; attempt++) {
    const errors = validateFlow(flow);
    if (errors.length === 0) return flow;
    flow = flow.__needsRegeneration
      ? await generateFlow(prompt, { feedback: errors })
      : repairFlow(flow, errors);
  }
  throw new CompileError('Unable to produce a valid Flow');
}

```

3.6 Properties of the Compiler

- **Bounded.** Retries are capped, so compilation always terminates.
- **Deterministic where it matters.** Probabilistic generation is confined to proposing a candidate. Acceptance is always decided by deterministic code.
- **Observable.** Structured error codes make failures diagnosable and testable.
- **Safe by default.** Nothing reaches the runtime until `validateFlow` returns an empty error list.

4. Deterministic Runtime & Security: The Engine

4.1 The Core Security Invariant

The runtime never executes model-written code.

There is no `eval`, no `new Function`, and no dynamic script injection anywhere in the execution path. The model's output is *data*: a graph description. The runtime is a fixed interpreter that dispatches on a node's `type` field to pre-written, audited handlers. This eliminates the largest risk class in LLM-driven automation, prompt-induced arbitrary code execution, by making it structurally impossible rather than heuristically filtered.

4.2 The Closed Node Set

The runtime recognizes exactly five node types:

Node Type	Responsibility
<code>trigger</code>	Entry point that initiates a run
<code>http-get</code>	Performs an allowlisted, keyless HTTP GET
<code>transform</code>	Reshapes or extracts fields from upstream data
<code>filter</code>	Conditionally retains or discards data
<code>output</code>	Renders final results to the UI

Any node outside this set is rejected by the validator (Section 3) and, as defense in depth, by the runtime dispatcher itself.

4.3 Graph Execution

`runFlow(flow)` computes a topological order over the DAG and invokes each node's handler, passing upstream output as input.

```

async function runFlow(flow) {
  const errors = validateFlow(flow);
  if (errors.length) throw new FlowError('Refusing to run invalid flow', errors);

  const order = topologicalSort(flow);
  const results = new Map();

  for (const node of order) {
    const inputs = incomingEdges(flow, node.id).map(e => results.get(e.from));
    let value;

    switch (node.type) {
      case 'trigger': value = { startedAt: Date.now() }; break;
      case 'http-get': value = await executeHttpGet(node.params); break;
      case 'transform': value = applyTransform(node.params, inputs[0]); break;
      case 'filter': value = applyFilter(node.params, inputs[0]); break;
      case 'output': renderOutput(node.params, inputs[0]); break;
      default:
        // Unreachable after validation; fail closed regardless
        throw new FlowError(`Unsupported node type: ${node.type}`);
    }
    results.set(node.id, value);
  }
  return results;
}

```

Note the **fail-closed default branch**: even if validation were somehow bypassed, an unrecognized type halts execution instead of falling through.

4.4 Network Egress: The Keyless API Allowlist

The only node capable of network I/O is **http-get**, and it is restricted to a fixed allowlist of keyless public APIs:

- **Open-Meteo**: weather and forecast data
- **Hacker News**: technology news and discussion
- **Frankfurter**: currency exchange rates

```

const ALLOWED_ORIGINS = new Set([
  'https://api.open-meteo.com',
  'https://hacker-news.firebaseio.com',
  'https://api.frankfurter.app'
]);

async function executeHttpGet(params) {
  let url;
  try {
    url = new URL(params.url);
  } catch {
    throw new FlowError('Invalid URL');
  }

  if (url.protocol !== 'https:' || !ALLOWED_ORIGINS.has(url.origin)) {
    throw new FlowError(`Blocked: ${url.origin} is not on the allowlist`);
  }

  const response = await fetch(url.href, { method: 'GET', credentials: 'omit' });
  if (!response.ok) throw new FlowError(`Upstream error: ${response.status}`);
  return response.json();
}

```

This design neutralizes several threat categories:

- **SSRF.** A malicious or hallucinated Flow cannot direct requests toward internal network addresses, localhost services, or cloud metadata endpoints, because only exact, pre-approved origins pass the check.
- **Data exfiltration.** `http-get` cannot be pointed at attacker-controlled hosts, and no node type sends data outward.
- **Credential leakage.** Requests use `credentials: 'omit'`, and the allowlisted APIs require no keys, so there is nothing sensitive to leak.

Origin matching uses parsed `URL` objects rather than string prefixes, which defeats classic bypasses such as `https://api.open-meteo.com.evil.example`.

4.5 XSS Prevention: Treating API Data as Hostile

Even allowlisted APIs return data ORANGE does not control. Hacker News titles, for example, are user-submitted content. ORANGE therefore treats every upstream response as untrusted. The `output` node renders exclusively through `textContent`, never `innerHTML`:

```
function renderOutput(params, data) {
  const container = document.getElementById('output');

  const row = document.createElement('div');
  row.className = 'output-row';

  const label = document.createElement('strong');
  label.textContent = params.label ?? 'Result';

  const body = document.createElement('span');
  body.textContent = typeof data === 'string' ? data : JSON.stringify(data, null, 2);

  row.append(label, body);
  container.appendChild(row);
}
```

Because `textContent` assigns a string as a literal text node, markup such as `<img src=x onerror=...>` in an API response is displayed as inert characters and is never parsed as HTML. The HTML parser is never invoked on untrusted data.

4.6 Defense-in-Depth Summary

Layer	Control	Threat Mitigated
Compiler	Schema-constrained generation	Malformed or off-spec output
Validator	<code>validateFlow</code> closed type set	Hallucinated or injected node types
Interpreter	Fixed handlers, fail-closed dispatch	Arbitrary code execution
Network	Origin allowlist, HTTPS only	SSRF, exfiltration
Request policy	<code>credentials: 'omit'</code> , keyless APIs	Credential leakage
Rendering	<code>textContent</code> only	XSS via untrusted API data

5. Ephemeral State Hydration

5.1 Sharing Without a Database

Conventional workflow tools persist definitions in a server-side datastore and share them by ID. ORANGE inverts this. Because a Flow is a compact, self-describing JSON document, the **URL itself is the database**. A workflow is serialized into the URL fragment, and opening that URL reconstructs the workflow entirely on the client.

5.2 Serialization Pipeline

```
Flow (object) --> JSON.stringify --> UTF-8 encode --> Base64 (btoa) --> #flow=<payload>
```

5.3 Writing State to the URL

```
function updateShareURL() {
  const json = JSON.stringify(currentFlow);

  // btoa() operates on Latin-1; encode to UTF-8 bytes first for Unicode safety
  const bytes = new TextEncoder().encode(json);
  let binary = '';
  bytes.forEach(b => (binary += String.fromCharCode(b)));
  const encoded = btoa(binary);

  const url = `${location.origin}${location.pathname}#flow=${encoded}`;
  window.history.replaceState(null, '', url);
}
```

`window.history.replaceState()` updates the address bar **without** a page reload and **without** polluting the browser's back-button history with an entry for every edit. The user can copy the URL at any moment and share a faithful snapshot of the current workflow.

5.4 Hydrating State from the URL

```
function hydrateFromURL() {
  const match = location.hash.match(/^#flow=(.+)$/);
  if (!match) return null;

  try {
    const binary = atob(match[1]);
    const bytes = Uint8Array.from(binary, c => c.charCodeAt(0));
    const flow = JSON.parse(new TextDecoder().decode(bytes));

    // Critical: shared state is untrusted input
    const errors = validateFlow(flow);
    return errors.length === 0 ? flow : null;
  } catch {
    return null;
  }
}
```

5.5 Security Posture of Shared Flows

A URL fragment is attacker-controllable, so ORANGE treats it with the same suspicion as model output. A hydrated Flow passes through the **same `validateFlow` gate** before it can execute. A crafted link can therefore do no more than any legitimate Flow: it can invoke only the five node types and reach only the

allowlisted APIs. Additional properties of fragment-based sharing:

- **Privacy-preserving.** URL fragments are not transmitted to servers in HTTP requests. A shared workflow never touches infrastructure, even the static host.
- **Zero persistence cost.** There is no storage layer to provision, secure, migrate, or pay for.
- **Self-contained and portable.** A link carries the complete workflow, and it works anywhere ORANGE is served.
- **Ephemeral by design.** State exists only where the link exists, which suits lightweight automation that does not need durable server-side records.

Conclusion

ORANGE demonstrates that the prevailing assumption of AI infrastructure, that capable automation requires a cloud backend, is a choice and not a necessity. By combining WebGPU-accelerated on-device inference, schema-constrained decoding, a deterministic closed-world runtime, and URL-native state, ORANGE delivers a complete automation studio from a set of static files.

The design philosophy generalizes beyond this system. Probabilistic models are best used to *propose*, while deterministic systems decide what is *accepted* and *executed*. When a small model is confined to a typed schema and wrapped in a validator, it becomes a reliable compiler front-end. When a runtime interprets a closed instruction set, it becomes safe regardless of what the model says. When state lives in a URL, infrastructure disappears.

As WebGPU adoption matures, on-device models improve, and browsers continue to expand their compute capabilities, the economics and privacy advantages of this approach will only deepen. The browser is no longer merely a client for AI services. It is becoming the platform on which AI runs.

Describe it. Orange compiles it.