VISION DOCUMENT

Claw: The Governance-First Browser Agent

Evolving the MCP Risk-Aware Summarizer into a production-grade, OPA-hardened platform for secure browser-to-model context injection.

Author: **Saatvix.**

Date: February 2026
Classification: Internal Strategy — Portfolio Reference

Contents

Executive Summary

The Core Thesis
Every browser tab is an unguarded context window into the enterprise. Claw makes it governable. We intercept, evaluate, and policy-gate browser content before it ever reaches an LLM, transforming uncontrolled browser-to-model data flow into an auditable, compliant, zero-trust pipeline.

The MCP Risk-Aware Summarizer demonstrated a working browser-to-model bridge: a Firefox extension that extracts page content, routes it through a local MCP server, and returns a risk-scored analysis from Claude. That prototype validated the core mechanics. This document proposes the next evolution: integrating **Open Policy Agent (OPA)** as a Policy Enforcement Point (PEP) to transform the bridge into a governance-first platform we're calling **Claw**.

Claw addresses three audiences simultaneously. For the **Product Manager**, it's a zero-config governance tool with pre-built policy packs. For the **Researcher**, it's a declarative framework for prompt-level security experiments. For the **Security Engineer**, it's a hardened pipeline with PII masking, domain guardrails, and prompt injection detection. The unified architecture serves all three without compromise.

This document delivers the unified vision: the API-first architecture, the security logic flow, the distribution strategy, and a phased roadmap to production. It also addresses the dual-audience challenge of developer adoption and CTO assurance through intentional documentation design.

Unified Architecture

Current State: The Validated Prototype

The existing system follows a linear extraction-analysis pipeline. The Firefox extension's content script uses DOM heuristics to extract readable text, the popup sends that payload via HTTP POST to a FastAPI bridge on localhost:8787, and the bridge invokes the MCP tool that calls Claude with a Risk Auditor system prompt. Claude returns structured JSON with a three-bullet summary, a 0–10 risk score, and severity-flagged safety assessments.

This architecture works, but it has a critical gap: **there is no policy evaluation between content extraction and model submission**. Every extracted page reaches Claude regardless of its domain, content classification, or the presence of sensitive data. That's the gap OPA fills.

Target State: The OPA-Gated Pipeline

The evolved architecture inserts OPA as an inline policy enforcement layer between the extension and the model. The data flow becomes:

```
Firefox Extension
  │ DOM extraction + metadata
  ▼
Claw Gateway (HTTP Bridge)
  │
  ├── Stage 1: Pre-Processing
  │    └ PII Scanner → Redact/Mask
  │
  ├── Stage 2: OPA Policy Evaluation
  │    ├ Domain allowlist/blocklist
  │    ├ Content classification gate
  │    ├ Prompt injection detection
  │    └ Data volume / rate limiting
  │
  ├── Stage 3: Context Assembly
  │    └ Policy-approved content → MCP Tool
  │
  └── Stage 4: Model Inference
       └ Claude API → Risk-Aware Response

  Every stage emits structured audit logs.
```

The key architectural principle is that **OPA never sees the model and the model never sees unfiltered content**. OPA operates on metadata and content features (domain, classification, token count, PII scan results) rather than raw text. This separation means policies remain lightweight and the model receives only approved, sanitized context.

The Four-Stage Pipeline

Stage 1: Pre-Processing

Before any policy evaluation, the raw extracted text passes through a PII scanning and masking layer. This uses a combination of regex patterns for structured data (SSNs, credit cards, emails, phone numbers) and a lightweight NER model for names and addresses. Detected PII is either redacted (replaced with [PII_TYPE]) or masked (partially obscured), depending on the active policy pack. The scan results become metadata that feeds into Stage 2.

Stage 2: OPA Policy Evaluation

This is the core innovation. The gateway constructs an OPA input document from the request metadata and pre-processing results, then queries the OPA engine for a policy decision. The input document looks like this:

```
{
    "input": {
      "url": "https://internal.corp.com/finance/q3-report",
      "domain": "internal.corp.com",
      "title": "Q3 Financial Report - Confidential",
      "content_features": {
        "word_count": 4200,
        "language": "en",
        "pii_detected": {
          "ssn": 0, "email": 3, "phone": 1, "name": 12
        },
        "classification_signals": ["financial", "confidential"]
      },
      "user": {
        "role": "analyst",
        "department": "finance"
      },
      "policy_pack": "finance"
    }
  }
```

OPA evaluates this against the active Rego policy bundle and returns a decision with three possible outcomes: **allow** (content proceeds to the model with any specified transformations), **allow_with_modifications** (content proceeds but with mandatory redactions or truncations), or **deny** (content is blocked entirely and the user receives an explanation).

Stage 3: Context Assembly

Approved content is assembled into the final MCP tool invocation. If the policy decision included modifications (additional redactions, context pruning, or field removal), those are applied here. The assembled payload is immutable once constructed and its hash is logged for audit purposes.

Stage 4: Model Inference

The existing Claude integration executes against the sanitized, policy-approved context. The Risk Auditor system prompt and structured JSON output remain unchanged. The model's response is also logged against the request hash for end-to-end traceability.

OPA Policy Design

Policy Pack Architecture

Rather than requiring users to write Rego from scratch, Claw ships with curated **Policy Packs** that encode common governance postures. Each pack is a versioned bundle of Rego files with accompanying metadata. Users select a pack at installation time and can customize it later.

Pack	Domain Rules	PII Handling	Content Gate	Use Case
Standard	Public domains only	Mask emails/phones	None	Personal browsing, general research
Finance	Allowlisted + internal	Full redaction, block SSN/CC	Block confidential-tagged	Financial services compliance
Strict	Explicit allowlist only	Block all PII categories	Require classification < secret	Defense, government, healthcare
Research	All domains, log only	Detect but pass through	Warn but allow	Security research, testing

Core Rego Patterns

The following illustrates the domain guardrail policy. Note how the logic is purely declarative and operates on metadata, never on raw content:

```
package claw.domain

import rego.v1

# Default: deny all domains not explicitly allowed
default allow := false

# Allow if domain is in the public allowlist
allow if {
    some domain in data.allowlists.public_domains
    input.domain == domain
}

# Allow internal domains only for authorized roles
allow if {
    endswith(input.domain, ".corp.com")
    input.user.department in data.internal_access[input.user.role]
}

# Always deny known exfiltration targets
deny if {
    some domain in data.blocklists.exfiltration_targets
    input.domain == domain
}
```

```
# Deny overrides allow
decision := "deny" if { deny }
decision := "allow" if { allow; not deny }
decision := "deny" if { not allow }
```

Prompt Injection Detection via OPA

One of the most novel applications of OPA in this architecture is using it as a prompt injection pre-filter. Rather than asking Claude to self-detect injections (which is inherently unreliable), we evaluate content features against known injection patterns **before** the content reaches the model. The Rego rules check for injection markers: role-override phrases, instruction delimiters, encoded payloads, and statistical anomalies like unusually high directive-to-content ratios.

This is not a complete solution—adversarial prompt injection is an open research problem—but it raises the cost of attack significantly by creating a policy layer the attacker cannot influence, because OPA never sees the model and the model never sees OPA's decisions.

Audit Trail

Every OPA decision is logged as a structured event with the following fields: timestamp, request hash, policy version, decision outcome, matched rules, applied modifications, and the user context. These logs are append-only and designed to feed into SIEM systems. For compliance-heavy environments, the audit trail provides the evidence chain: what content was evaluated, what policy was applied, what was modified or blocked, and what ultimately reached the model.

API-First Design

> **Design Philosophy: The Stripe Analogy**
> Stripe made payments simple by hiding the complexity behind a clean API. Claw does the same for browser context governance. A developer should be able to POST browser content and receive a policy-gated, risk-scored analysis in a single call. The governance layer is invisible when you want it to be, and fully observable when you need it to be.

OpenAPI Surface

The API is organized around three resource groups: **Context** (submit and analyze browser content), **Policy** (manage and inspect OPA policies), and **Audit** (query decision logs). The primary endpoint remains POST /analyze, but it now returns enriched metadata about the policy decisions that shaped the response.

Core Endpoints

Method	Path	Description	Auth
POST	/v1/analyze	Submit browser content for policy-gated analysis	API Key
GET	/v1/analyze/{request_id}	Retrieve a previous analysis by ID	API Key
GET	/v1/policy/active	Get the currently active policy pack and its rules	Admin
PUT	/v1/policy/active	Switch the active policy pack	Admin
GET	/v1/policy/packs	List all available policy packs	API Key
POST	/v1/policy/evaluate	Dry-run policy evaluation without model inference	API Key
GET	/v1/audit/decisions	Query the policy decision audit log	Admin
GET	/v1/audit/decisions/{id}	Get detailed audit trail for a specific decision	Admin
GET	/v1/health	Server health, OPA status, and model availability	None

The /v1/analyze Response Envelope

The analyze endpoint returns a response that wraps the Claude analysis inside a governance context. This is the key design decision: every response tells you not just **what** the model said, but **what policies shaped what the model saw**.

```
{
  "request_id": "clw_req_7k9x2m",
  "status": "analyzed",
  "policy": {
    "pack": "finance",
    "version": "1.2.0",
    "decision": "allow_with_modifications",
    "modifications_applied": [
      "pii_redaction: 3 email addresses masked",
      "pii_redaction: 12 names replaced with [NAME]"
    ],
    "rules_evaluated": 14,
    "evaluation_ms": 3
  },
  "analysis": {
    "summary": [
      "The Q3 report shows 12% YoY revenue growth...",
      "Operating margins contracted by 2 points due to...",
      "Management guidance for Q4 projects acceleration in..."
    ],
    "risk_score": 2,
    "risk_rationale": "Factual financial reporting with proper...",
    "safety_flags": [
      { "severity": "ok", "message": "Content verified as..." }
    ]
  },
  "audit": {
    "content_hash": "sha256:a1b2c3...",
    "model": "claude-sonnet-4-5-20250514",
    "tokens_in": 1847,
    "tokens_out": 312,
    "latency_ms": 2340
  }
}
```

The /v1/policy/evaluate Endpoint (Dry Run)

This is the developer’s best friend. It accepts the same input as /v1/analyze but only runs the OPA evaluation—no model inference, no API cost. Developers can iterate on policy configurations by testing content against different packs and seeing exactly which rules fire, what modifications would be applied, and whether the content would be allowed or denied. This endpoint is also critical for CI/CD integration: policy changes can be tested against a corpus of sample inputs before deployment.

Distribution Strategy

The Problem with Python/pip

The current server requires Python, pip, and manual dependency installation. This is acceptable for developer prototyping but unacceptable for enterprise deployment. The friction points are: Python version conflicts, virtual environment management, pip dependency resolution failures, OPA binary installation, and the absence of a single-command startup experience. Each of these is a support ticket waiting to happen.

Proposed Solution: Go-Based Single Binary

The production distribution repackages the entire server stack into a single statically-linked Go binary. The choice of Go is deliberate:

- **Single binary, zero dependencies.** No Python, no pip, no virtual environments. Download, run, done.
- **OPA is Go-native.** The OPA engine itself is written in Go and can be embedded as a library (github.com/open-policy-agent/opa/rego), eliminating the need for a sidecar process.
- **Cross-compilation.** A single CI pipeline produces binaries for macOS (arm64/amd64), Linux (amd64), and Windows (amd64) with zero per-platform build complexity.
- **Embedded policy bundles.** Policy packs are compiled into the binary using Go's embed directive, so the default policies ship with the executable.
- **Native HTTP server.** Go's net/http replaces FastAPI/Uvicorn with better performance and no runtime overhead.

The Claude API integration would use the Anthropic Go SDK (or raw HTTP calls to the messages endpoint). The MCP tool definition remains identical; only the transport layer changes.

The One-Command Experience

```
# macOS / Linux
curl -fsSL https://get.clawagent.dev | sh

# Or download directly
brew install sahasra/tap/claw

# Start with defaults (Standard policy pack)
claw serve --api-key $ANTHROPIC_API_KEY

# Start with Finance policy pack
claw serve --api-key $ANTHROPIC_API_KEY --policy finance

# Health check
curl http://localhost:8787/v1/health
```

Extension Distribution

The Firefox extension ships through Mozilla's Add-ons store (AMO) for public distribution, with a .xpi sideload option for enterprises that manage browser configurations centrally. The extension auto-detects the local Claw server on startup and displays connection status in the popup. For Chrome/Edge support, the same codebase ships as a Manifest V3 Chrome extension with minimal browser API shims.

Security Architecture

Threat Model

The threat model centers on a single adversary: **sensitive browser context reaching an LLM without authorization**. This encompasses several attack vectors and failure modes:

Threat	Vector	Mitigation	OPA Role
PII Exfiltration	Extracted DOM contains SSNs, CC numbers	Pre-processing PII scanner + policy gate	Enforce redaction/deny based on PII counts
Unauthorized Domain	User scans internal/classified page	Domain allowlist/blocklist in Rego	Deny requests from non-approved domains
Prompt Injection	Malicious page content manipulates Claude	Pattern detection on content features	Flag high-risk content signatures pre-model
Data Volume Attack	Adversary triggers bulk extraction	Rate limiting + token budget per session	Enforce rate/volume limits per user/session
Policy Bypass	Attacker modifies extension to skip OPA	Server-side enforcement; extension is untrusted	All policy logic runs server-side, never client
Audit Tampering	Admin deletes incriminating logs	Append-only log with cryptographic chaining	Decision logs include hash chain verification

Zero-Trust Principle

The architecture treats the Firefox extension as an **untrusted input source**. All security enforcement happens server-side in the Claw gateway. The extension is merely a content extraction and display layer. This means a compromised or modified extension cannot bypass policy: even if an attacker patches the extension to skip client-side checks, the server independently validates every request against OPA policies before allowing model inference.

This is the key insight that differentiates Claw from other browser automation tools: the security boundary is at the gateway, not at the browser.

Defense in Depth Layers

Layer	Component	What It Catches
L1: Transport	TLS + localhost binding	Network interception, remote access attempts
L2: Input	Schema validation + size limits	Malformed requests, payload bombs
L3: Content	PII scanner + classification	Sensitive data, document classification signals
L4: Policy	OPA Rego evaluation	Domain violations, unauthorized access, injection patterns
L5: Assembly	Context sanitization + hashing	Residual PII, payload tampering between stages
L6: Model	Claude Risk Auditor prompt	Content-level risk, bias, misinformation
L7: Audit	Structured decision logging	Post-hoc compliance review, forensic analysis

Documentation Strategy

The Dual-Audience Problem

Developer documentation must be approachable enough to get a prototype running in 5 minutes, while simultaneously providing the depth and assurance that a CTO or CISO needs to approve deployment. These audiences read documents differently: developers scan for code snippets and quickstart guides; executives scan for architecture diagrams, compliance coverage, and risk mitigation. The documentation must serve both without diluting either.

Developer-Facing Documentation

The developer documentation follows the Stripe/Vercel model: a left-nav reference site with interactive examples. The structure is built on three principles.

Principle 1: Time-to-first-request under 5 minutes. The landing page shows a curl command that calls the /v1/analyze endpoint with a sample payload and displays the response. No signup, no configuration, no preamble. The developer sees the value proposition in a single terminal command.

Principle 2: Progressive disclosure. The quickstart deliberately hides OPA, policy packs, and audit features. These appear only when the developer clicks into the Governance section. This prevents the security layer from creating adoption friction.

Principle 3: Copy-paste completeness. Every code example is a complete, runnable snippet. No "... (configuration omitted)" ellipses. No assumptions about the developer's environment beyond having the Claw binary and an API key.

CTO/CISO-Facing Documentation

The executive documentation is a separate path on the same site, accessible via a "For Security Teams" header link. It leads with the threat model rather than the API, and is organized around compliance questions:

- **What data leaves the browser?** Full content extraction pipeline documentation with redaction examples.
- **What data reaches the model?** OPA policy evaluation walkthrough showing what gets blocked and modified.
- **Where are the audit logs?** Decision log schema, retention policies, and SIEM integration guides.
- **What is the blast radius of a compromise?** Localhost binding, no persistent storage, stateless architecture.
- **What compliance frameworks does this support?** Mappings to SOC 2, GDPR Article 25 (privacy by design), and NIST 800-53.

The critical design choice is that the executive section **never simplifies the technical details**. CISOs distrust marketing language. Instead, it presents the same architecture and threat model that engineers see, but organized around the questions executives actually ask.

Implementation Roadmap

Phase 1: OPA Integration (Weeks 1–4)

- Embed OPA engine into the existing Python server as a sidecar process
- Implement the four-stage pipeline (pre-processing, OPA evaluation, context assembly, inference)
- Build the Standard and Finance policy packs in Rego
- Add structured audit logging with request hash chaining
- Implement PII scanning using regex patterns + spaCy NER
- Ship /v1/policy/evaluate dry-run endpoint
- Deliverable: Working OPA-gated server with policy packs, still Python-based

Phase 2: Go Rewrite + Distribution (Weeks 5–8)

- Rewrite the HTTP bridge and pipeline orchestration in Go
- Embed OPA as a Go library (eliminate sidecar)
- Embed default policy bundles using Go embed
- Implement the full OpenAPI v1 surface (analyze, policy, audit)
- Build cross-platform CI/CD pipeline (macOS arm64/amd64, Linux, Windows)
- Create the one-command installer script and Homebrew tap
- Deliverable: Single-binary Claw server with zero external dependencies

Phase 3: Extension Hardening + Store Distribution (Weeks 9–12)

- Submit Firefox extension to AMO for review and public listing
- Port extension to Chrome/Edge Manifest V3
- Add extension auto-discovery of local Claw server
- Implement extension-side connection status and policy pack display
- Build .xpi/.crx sideload packages for enterprise MDM deployment
- Deliverable: Production Firefox and Chrome extensions on public stores

Phase 4: Documentation + Enterprise Features (Weeks 13–16)

- Build developer documentation site (Mintlify or Nextra-based)
- Create interactive API playground with sample payloads
- Write the CTO/CISO security documentation path
- Add SOC 2 and GDPR compliance mapping documentation
- Implement Strict policy pack for government/defense use cases
- Add SIEM export integration (Splunk, Elastic, Datadog) for audit logs
- Deliverable: Complete documentation site, enterprise-ready policy pack, SIEM integration

Success Metrics

Metric	Target	Measurement
Time-to-first-analysis	< 5 minutes from download	Timed user testing with clean machine
Policy evaluation latency	< 10ms p99	OPA decision log timestamps

Metric	Target	Measurement
End-to-end latency	< 4 seconds (incl. Claude API)	Request-to-response timing
PII detection recall	> 95% for structured PII types	Test corpus with labeled PII
Audit log completeness	100% of decisions logged	Log count vs request count comparison
Extension store rating	> 4.5 stars on AMO	Mozilla Add-ons store metrics
Documentation coverage	100% of API endpoints documented	OpenAPI spec vs docs site diff

Closing: The Competitive Moat

Browser automation tools exist. AI summarizers exist. Policy engines exist. What does not exist is a **single product that combines all three with zero-config simplicity**. The competitive moat of Claw is the integration itself: the fact that a developer can download a single binary, install a browser extension, and immediately have a policy-gated, risk-scored, audit-trailed browser-to-model pipeline. No YAML configuration files, no Docker containers, no infrastructure provisioning.

The deeper moat is the OPA integration. Once an enterprise deploys custom Rego policies, those policies encode institutional knowledge about data classification, access control, and compliance requirements. That policy corpus becomes increasingly valuable and increasingly difficult to replicate with a competing tool. The switching cost is not the binary—it's the policies.

Claw is not a browser extension. It is not an MCP server. It is a **governance primitive for the age of browser-connected AI agents**. Every enterprise will eventually need to answer the question: "what browser context is reaching our AI models, and who authorized it?" Claw provides that answer, today.

End of Document