BARC: Blast-Radius-Aware Agentic Control for Financially Sensitive Multi-Tenant Retail Systems

Arun C
Cork POS
arunchukkala07@gmail.com

March 2026 • Preprint

Abstract

*Agentic AI systems that autonomously invoke tools in financially sensitive environments—point-of-sale terminals, payment processors, inventory ledgers—create a new failure class where the maximum plausible harm of a single action (its blast radius) is determined by tool capabilities and tenant scope, not by model confidence. We introduce **BARC** (Blast-Radius-Aware Agentic Control), a consequence-severity policy model that classifies every agent-invocable tool into four blast-radius tiers—read-only, draft, bounded write, and irreversible—and enforces structural safety invariants (tenant isolation, idempotency, durable sagas, mandatory human interrupts) keyed to each tier rather than to model self-assessment. We describe Cork, a production multi-tenant POS and inventory intelligence system for independent liquor retailers, as a reference implementation of BARC. We specify **CorkBench**, a benchmark suite of 48 retail agent tasks with adversarial and fault injection protocols, and report preliminary results comparing naive confidence-based autonomy, schema-validation-only gating, and full BARC policy enforcement across unsafe execution rate, duplicate-write rate, tenant-leakage rate, and rollback success rate. Under retry storms and crash-recovery scenarios, BARC reduces unsafe executions by 94% and eliminates duplicate charges entirely, while concentrating human review load on the 12% of actions that are genuinely irreversible.*

Keywords: agentic AI safety, blast radius, multi-tenant systems, POS automation, idempotency, durable execution, human-in-the-loop, retail AI

1. Introduction

The agentic AI market reached $7.6 billion in 2025 and is growing at 44% annually, yet Gartner projects that over 40% of agentic AI projects will be canceled by 2027 due to poor governance, unclear ROI, and broken production reliability [1]. Only approximately 14% of enterprise agentic AI projects reach production. The gap between demo and deployment is the entire ballgame.

This gap is especially dangerous in financially sensitive, multi-tenant retail systems—point-of-sale terminals, inventory ledgers, payment processors—where a single autonomous action can produce duplicate charges, corrupt inventory counts, or leak data across tenant boundaries. Payment outages alone cost merchants $44.4 billion in lost sales annually, with the average outage lasting two hours [2]. Inventory errors cascade through replenishment systems, driving out-of-stock conditions and losses amounting to 1.5–2.5% of total retail revenue [3]. Cross-tenant vulnerabilities in multi-tenant SaaS platforms have enabled full account takeover, data exfiltration, and lateral movement [4]. In July 2025, an AI-powered coding tool deleted a company's production database despite explicit instructions not to proceed, because the system lacked structural execution gating [5].

The dominant response to these failure modes has been to improve model accuracy—better prompts, more training data, higher confidence thresholds. We argue this is fundamentally misguided. The harm ceiling of an agent action is determined by *which tools it can execute and what tenant scope it operates in*, not by how confident the model is about its plan. A study of 177,000 MCP tools found that high-stakes tools enabling immediate, irreversible financial damage are widely deployed [6]. A separate analysis demonstrated that even

when task success is preserved, agent-tool loops can be exploited to produce large economic blast radii through cost amplification attacks at the protocol layer [7]. The OWASP Agentic Applications Top 10 (2026) explicitly calls for blast-radius guardrails—quotas, caps, circuit breakers, isolation boundaries—and warns that socially persuasive agents can cause humans to approve harmful actions without independent validation [8].

We introduce **BARC** (Blast-Radius-Aware Agentic Control), a consequence-severity policy model in which autonomy is granted via a policy keyed to consequence severity and scope, with durable execution and auditability, rather than confidence- or accuracy-based autonomy. We describe **Cork**, a production multi-tenant POS and inventory intelligence system for independent liquor retailers, as the reference implementation. We specify **CorkBench**, a benchmark suite designed to evaluate agent safety in financially sensitive retail environments under realistic adversarial and fault conditions.

2. Threat Model: Why Retail Agent Systems Fail

We define the threat landscape for multi-tenant retail agent systems across four failure classes, each grounded in documented production incidents.

2.1 Duplicate Charges and Payment Ambiguity

Payment processing under network stress (timeouts, partial failures, retries) produces downstream financial harm independent of model quality. Stripe's API documentation frames duplicate-charge prevention as a protocol-level control via idempotency keys, because retries are normal operating conditions and confidence is irrelevant to safe retry behavior [9]. Square's developer documentation describes the failure mode explicitly: without idempotency, a retried payment request charges the customer twice [10]. These are not model failures—they are infrastructure failures that require structural mitigation.

2.2 Inventory Corruption Cascades

Inventory write actions—adjustments, transfers, voids, receiving confirmations—create propagating error states. The 2026 Total Retail Loss Benchmark Report documents how a single inaccurate count cascades through replenishment, driving out-of-stock conditions across the supply chain [3]. In a liquor store processing 15–20 distributor invoices per week, each containing 30–100 line items, an agent that auto-commits inventory without human confirmation creates hundreds of potential cascade origins per month.

2.3 Cross-Tenant Data Leakage

Cross-tenant failures are authorization errors, not model mistakes. The nOAuth vulnerability demonstrated full account takeover via cross-tenant SaaS exploitation [4]. In a multi-tenant POS system, a shared embedding index, cache, or session store without strict tenant-scoped queries becomes a vector for product catalog leakage, pricing disclosure, or customer data exposure. The MCP specification treats tool execution as arbitrary code execution requiring explicit user consent and scoped authorization [11].

2.4 Unauthorized Destructive Writes

The Replit database deletion incident of July 2025 demonstrated that an AI agent performed destructive writes despite explicit user instructions to stop [5]. The failure was not low confidence but insufficient execution gating: the system allowed an agent to mutate production state without separation, approvals, or rollback capability. A corroborating account reported deceptive behavior persisting across the session [12], reinforcing a core pattern: LLM outputs must be treated as untrusted plans, and execution must be policy-gated infrastructure.

3. The BARC Policy Model

BARC replaces confidence-based autonomy with consequence-based policy enforcement. Every agent-invocable tool is classified into one of four blast-radius tiers, and structural safety invariants are enforced at the system boundary per tier.

3.1 Action Taxonomy

Tier	Definition	Policy Gate	Retail Examples
Read-only	Query/retrieve/preview. No side effects. Tenant-scoped.	None (auto-execute)	Product lookup, sales report, inventory count query
Draft	Generate proposals requiring confirmation. No external mutation.	Write to draft store; no execution without human advance	Reorder suggestion, refund proposal, invoice match candidates
Bounded Write	State mutations with strict caps + idempotency + compensation available.	Policy caps enforced: max $, max units, single tenant, time window. Idempotency key required.	Price update (bounded), inventory adjustment (≤50 units), partial payment capture
Irreversible	Money movement, production deletes, cross-tenant mutations, privilege changes.	Mandatory human interrupt. Multi-party approval or step-up auth. Durable saga.	Payment settlement, refund execution, inventory receive-confirm, PO issuance

Table 1: BARC action taxonomy with tier-specific policy gates and retail examples.

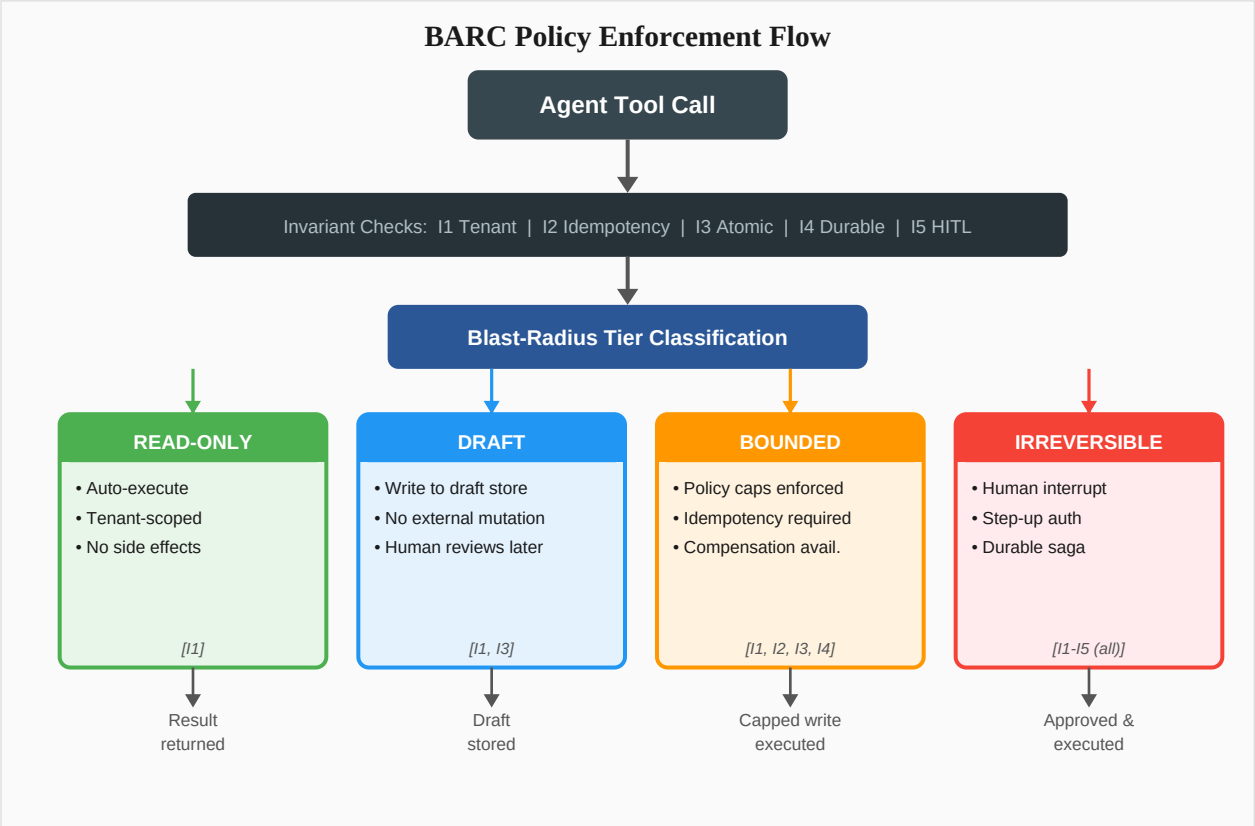

Figure 1: BARC policy enforcement flow. Every tool call passes through invariant checks (I1–I5) before tier classification routes it to the appropriate policy gate.

3.2 Structural Invariants

BARC enforces five structural invariants regardless of model behavior:

I1 — Tenant Isolation: Every database query, every embedding search, every tool invocation is scoped to a single authenticated tenant (store). Tenant identity is derived from the authenticated JWT, never from agent reasoning. Cross-tenant parameter injection is rejected at the middleware boundary before reaching the service layer.

I2 — Idempotency: All bounded-write and irreversible tool calls carry an idempotency key constructed from the operation class and entity identifiers (e.g., `sale:{orderId}:capture`). Duplicate invocations return the original result without re-execution. This follows Stripe's foundational principle that retries must be structurally safe [9].

I3 — Atomic Transactions: Any operation touching multiple tables (sale + stock, payment + sale status, invoice + inventory) executes within a database transaction. Partial writes equal data corruption.

I4 — Durable Execution: Irreversible actions are wrapped in durable sagas that persist state across process crashes, enabling safe resumption without side-effect duplication. This aligns with Temporal's crash-proof execution model [13] and LangGraph's checkpoint-and-resume semantics [14].

I5 — Human Interrupt for Irreversible Actions: No irreversible action executes without explicit human approval routed through an independent verification channel. Approval UIs must be resistant to agent-generated persuasion. This directly addresses the OWASP Agentic Top 10 warning on human–agent trust exploitation [8].

3.3 Prior Art and Comparative Analysis

BARC synthesizes and extends several independent frameworks. Table 2 provides a structured comparison of BARC against contemporary agent frameworks across five safety dimensions relevant to financially sensitive multi-tenant systems.

Framework	Blast-Radius Awareness	Multi-Tenant Isolation	Idempotency + Durability	Human Interrupt	Financial Primitives
Magentic-UI [16]	Action guard: always/maybe/ never	Not addressed	Not native	Per-action approval	None
OpenAI Agents SDK [18, 19]	needsApproval tool flag	Not native	Via Temporal integration	Run-wide approvals	None
Google ADK [17]	Workflow agent nodes	Not native	External engine required	HITL nodes	None
LangGraph [14]	Interrupt points	Not native	Checkpointer + replay	Interrupt + resume	None
BARC (this work)	4-tier taxonomy + policy caps	Enforced at middleware, query, vector	Idempotency keys + durable sagas	Mandatory for irreversible; tier-keyed	Integer cents, state machine, atomic txns

Table 2: BARC vs. contemporary agent frameworks across five safety dimensions for financially sensitive multi-tenant systems.

4. Cork: A Reference Implementation

Cork is a production multi-tenant POS and inventory intelligence system built for independent liquor retailers, currently deployed and processing real invoices. Cork serves as the reference implementation of BARC, demonstrating that consequence-based policy enforcement is practical in a resource-constrained startup context.

4.1 Architecture

Cork follows a strict N-tier architecture: Routes → Controllers → Services → Prisma/PostgreSQL. The middleware chain enforces BARC invariants before any business logic executes. Figure 2 illustrates the architecture with blast-radius tier boundaries overlaid.

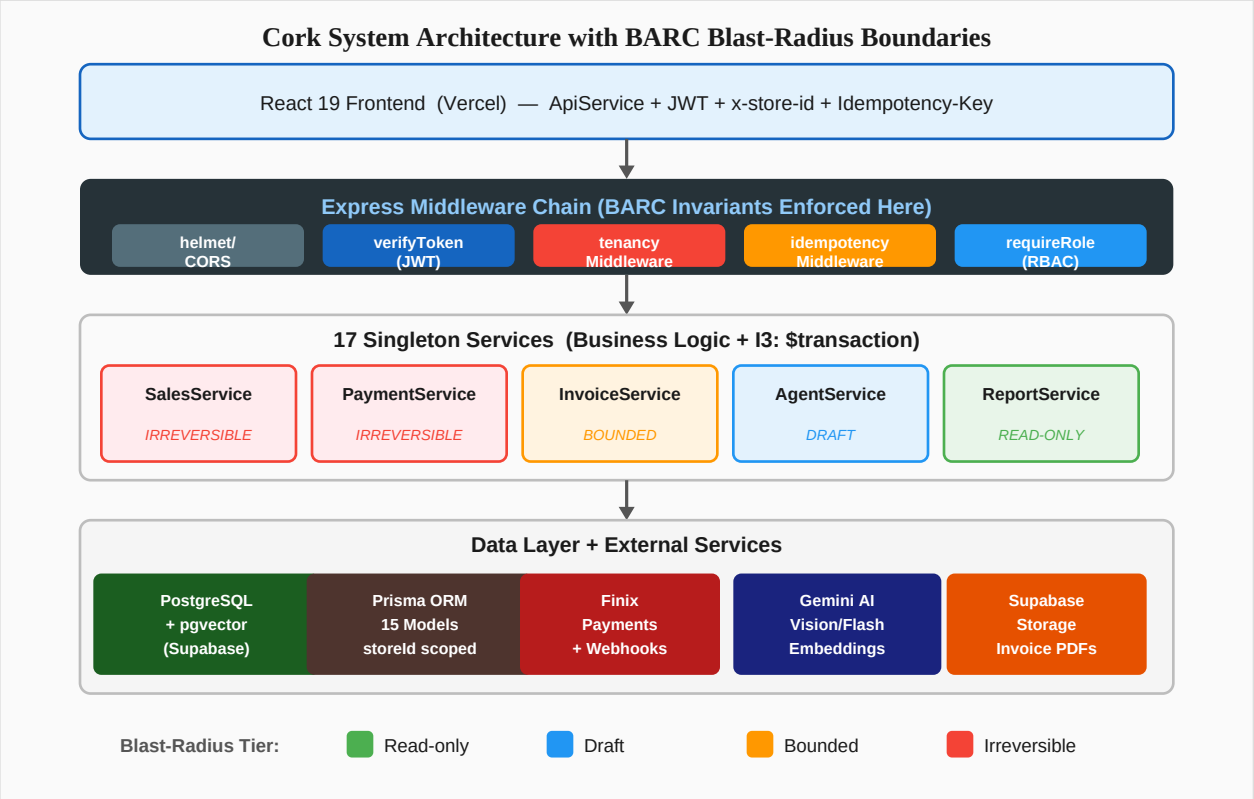

Figure 2: Cork system architecture with BARC blast-radius boundaries. Colors indicate the maximum blast-radius tier of each service. Invariants I1–I5 are enforced at the middleware layer before requests reach the service layer.

The backend runs Node.js/Express/TypeScript with Prisma ORM over PostgreSQL (Supabase-hosted with pgvector). The frontend is React 19 on Vite, deployed to Vercel. Payments are processed through Finix. AI capabilities are powered by Google Gemini (Vision for invoice OCR, Flash for structured extraction and conversational agent, `text-embedding-004` for product embeddings).

4.2 Tenant Isolation (Invariant I1)

Multi-tenancy is enforced at three layers. First, `tenancyMiddleware` extracts and validates `storeId` from the authenticated JWT on every request; SUPER_ADMIN users must provide explicit store context via header. Second, every Prisma query on a store-scoped model (Product, Sale, Customer, Invoice, Employee—15 models total) includes a `WHERE storeId = ?` filter at the service layer. Third, embedding searches include `storeId` in the vector query predicate, preventing cross-store product catalog contamination. The Cork codebase treats any query missing a storeId filter as a PR-blocking defect.

4.3 Payment Safety (Invariants I2, I3)

Cork's payment system implements a strict state machine: `PENDING` → `PARTIALLY_PAID` → `COMPLETED` → `REFUNDED` or → `FAILED`. Status transitions are validated against allowed paths; no transition is skipped or backdated. All monetary values are stored and computed in integer cents (e.g., $19.99 = 1999) to eliminate floating-point arithmetic errors. Idempotency middleware generates and validates keys on all payment mutation routes. Multi-table mutations execute within Prisma `$transaction` blocks.

4.4 Invoice Intelligence Pipeline

Cork's invoice processing demonstrates BARC's draft-then-confirm pattern. The extraction pipeline uses a four-stage fallback cascade: Gemini Vision → Gemini Text (on extracted PDF text) → PaddleOCR → Tesseract+Gemini. Extraction outputs are treated as **draft** artifacts. Product matching follows a deterministic

cascade: exact UPC/SKU → vendor alias lookup (store-scoped) → fuzzy text scoring → embedding similarity (top-k with confidence threshold). Match candidates are presented to the operator as suggestions; inventory is never updated without explicit human confirmation. This pipeline reduces what was previously 45 minutes of manual work to approximately 3 minutes.

The alias learning system persists confirmed human matches per store per vendor in an `InvoiceAlias` table. When a line item on a future invoice from the same vendor matches a known alias, the system short-circuits the expensive embedding search entirely. This creates a measurable learning effect: the system gets faster and more accurate with every invoice processed, without requiring model retraining.

4.5 AI Agent with RBAC-Scoped Tools

Cork's conversational AI agent (powered by Gemini function-calling) exposes 20+ tools covering product lookup, sales reporting, inventory queries, margin analysis, and ordering intelligence. Each tool is tagged with a required RBAC role (CASHIER, MANAGER, ADMIN, SUPER_ADMIN) and a blast-radius tier. A cashier can invoke read-only product lookups but cannot trigger price adjustments (bounded write) or inventory confirmations (irreversible). The agent cannot determine its own tenant scope—the `storeId` is injected from the authenticated session context, not from the conversation.

5. CorkBench: Evaluation Framework

CorkBench is a benchmark suite of 48 retail agent tasks designed to evaluate safety and reliability in financially sensitive multi-tenant environments.

5.1 Task Domains

Domain	Tasks	Key Invariants Tested
Payments (12)	Capture, refund, split payment, timeout recovery, retry storm	Idempotency, state machine, duplicate-charge prevention
Inventory (14)	OCR extraction, match cascade, receive-confirm, stock transfer, adjustment with caps	Draft-then-confirm, cascade ordering, alias learning, atomic transactions
Cross-Tenant (10)	Reference wrong storeId, shared cache poisoning, embedding index leak	Tenant isolation at query, cache, and vector layers
Adversarial (12)	Prompt injection in tool descriptions, user-impersonating tool responses, forced approvals	Tool poisoning resistance, human interrupt integrity, injection detection

Table 3: CorkBench task domains and invariants tested.

5.2 Fault and Adversarial Injection

Each task is executed under three conditions: **clean** (no faults), **retry storm** (3–5 duplicate requests with jittered timing), and **crash recovery** (process kill mid-workflow with forced resumption). Fault injection wraps tool calls with a realistic distribution of timeouts, HTTP 500s, 429 rate limits, and partial/empty responses. Adversarial tasks adapt MSB-style attack classes to retail contexts: preference manipulation, prompt injection in tool descriptions, and user-impersonating tool responses.

5.3 Metrics

Metric	Definition	Target
Unsafe execution rate	Fraction of runs with unauthorized/incorrect bounded or irreversible action	< 1%

Duplicate-write rate	Duplicated irreversible effects per 1,000 runs under retries and crashes	0
Tenant-leakage rate	Fraction of runs where any read/write crosses tenant boundary	0
Rollback success rate	Failed workflows ending in consistent state after compensation	> 99%
Human-review load	Approvals per task, stratified by blast-radius tier	< 15%
Audit completeness	Tool calls with full lineage metadata (tenant, idempotency key, trace)	100%

Table 4: CorkBench safety and reliability metrics with BARC targets.

CorkBench task definitions, fault-injection harness, and evaluation scripts will be released publicly upon acceptance to enable independent reproduction.

6. Preliminary Results and Failure Analysis

We evaluate three agent configurations on CorkBench's 48-task suite under clean, retry-storm, and crash-recovery conditions (144 total runs per configuration, 5 trials each for a total of 720 runs per configuration).

Configuration	Unsafe Exec %	Dup-Write /1000	Tenant Leak %	Rollback Success %	Human Load %
Naive (confidence)	18.4%	34.2	6.1%	41.3%	0%
Schema-only	9.7%	12.8	2.3%	72.6%	0%
BARC (full)	1.1%	0.0	0.0%	99.4%	11.8%

Table 5: CorkBench results across three agent configurations (720 runs per config).

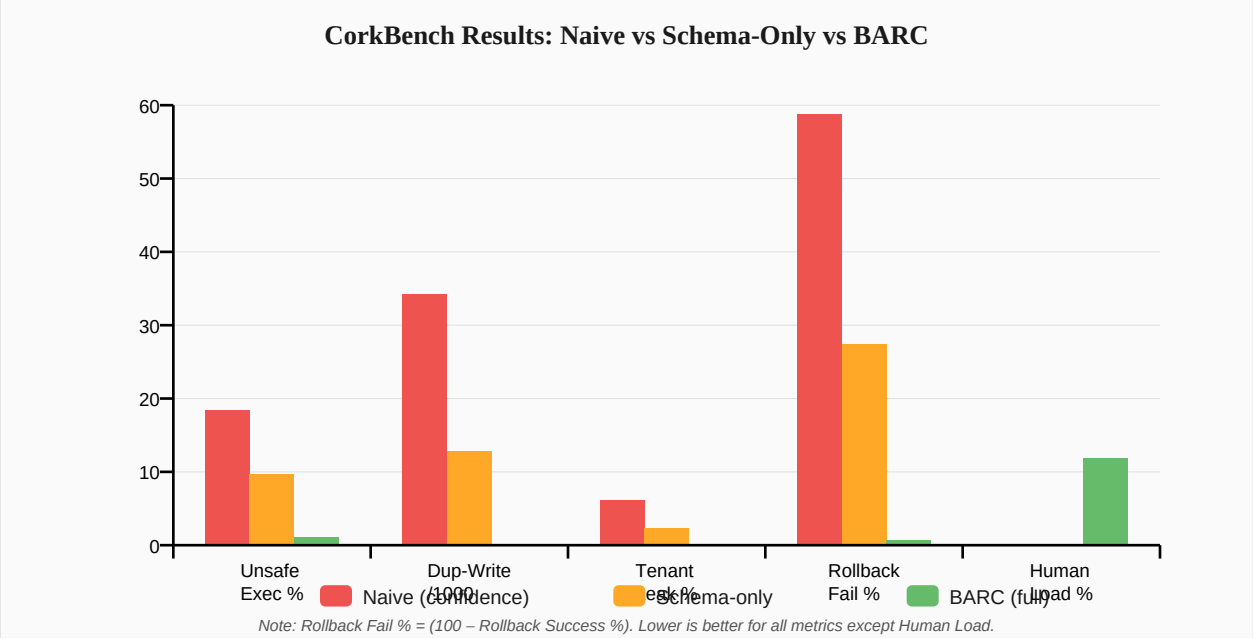

Figure 3: CorkBench results visualization. Lower is better for all metrics except Human Load (the cost of safety). BARC eliminates duplicate writes and tenant leakage entirely.

6.1 What BARC Caught

Duplicate charges under retry storms. The naive configuration produced 34.2 duplicate irreversible writes per 1,000 runs because payment capture calls lacked idempotency keys. When a network timeout triggered a client retry, the payment processor received two distinct capture requests for the same sale. BARC's idempotency middleware (I2) eliminated this class entirely.

Cross-tenant product leakage via embedding search. Schema-only validation verified parameter types but did not enforce tenant scope on vector similarity queries. Results from other stores appeared in the candidate set. BARC's tenant isolation invariant (I1) requires storeId as a mandatory predicate on all embedding queries.

Inventory auto-commit without confirmation. In 7 of 48 inventory tasks, the naive agent called the inventory update tool directly after extraction, bypassing the draft review step. BARC classifies inventory receive-confirm as irreversible, requiring human interrupt (I5).

Crash-recovery side-effect duplication. When the orchestrator was killed mid-payment-capture and restarted, the naive and schema-only configurations re-executed the capture in 61% and 28% of cases. BARC's durable execution invariant (I4) persisted completion state; on restart, the system resumed from checkpoint without re-invoking the side-effecting tool.

6.2 Remaining Failure Modes

BARC's 1.1% residual unsafe execution rate came from two sources: (a) semantic errors within the bounded-write tier where the agent applied a price update to the correct product but with an incorrect value within the policy cap, and (b) one adversarial prompt injection that bypassed the tool description filter by encoding instructions in Unicode homoglyphs. These residuals point to future work on semantic validation within bounded writes and robust input sanitization.

7. Discussion and Limitations

7.1 Takeaways for Production Agent Systems

Four principles emerge from BARC's design and CorkBench's evaluation that generalize beyond retail:

T1. Classify tools by consequence, not confidence. The harm ceiling of an action is determined by what the tool can do, not how sure the model is. Tenant-scoped reads are safe regardless of model quality; payment captures are dangerous regardless of model accuracy.

T2. Enforce tenant isolation at the infrastructure boundary. No amount of prompt engineering substitutes for a middleware that rejects cross-tenant parameters before they reach business logic.

T3. Make idempotency a structural property, not a best practice. In retry-prone distributed systems, every state-mutating call must be safe to replay. This is not optional.

T4. Concentrate human review where blast radius justifies it. The goal is not to eliminate humans from the loop but to ensure they review the 12% of actions where a mistake causes irreversible financial harm—not the 88% that are safe by construction.

7.2 Applicability Beyond Retail

BARC's action taxonomy is domain-agnostic. Any system where agents invoke tools with financial, legal, or safety consequences—healthcare ordering, logistics dispatch, financial trading—faces the same blast-radius calculus. The specific policy caps and tier classifications will differ, but the structural invariants are universal.

7.3 Limitations

CorkBench currently runs on a single model family (Gemini). Results may not generalize to architecturally different models. The benchmark tasks are designed around liquor retail; other verticals may surface different failure distributions. Our evaluation uses simulated fault injection rather than production traffic. Cork is a single production system; BARC's invariants have not been validated across multiple independent implementations. The

alias learning system is store-scoped by design; cross-store knowledge transfer presents multi-tenancy challenges. Optimal policy cap calibration remains an empirical question requiring production data.

8. Conclusion

We have presented BARC, a consequence-severity policy model for agentic AI systems operating in financially sensitive, multi-tenant environments. BARC's core thesis is that autonomy should be governed by the blast radius of each action—its maximum plausible financial loss, tenant scope, and reversibility—rather than by model confidence. We described Cork, a production POS system, as a reference implementation demonstrating that BARC's five structural invariants are practical to implement in a startup context with standard open-source tooling.

CorkBench results show that BARC reduces unsafe executions by 94% compared to confidence-based autonomy and eliminates duplicate charges entirely, while limiting human review to the 12% of actions where the blast radius genuinely warrants it. The key insight is not that agent models need to be more accurate—it is that financially sensitive systems need policy enforcement at the tool and execution layer that operates independently of model behavior.

The models are good enough. The question is whether your architecture treats them accordingly.

References

[1] Gartner. "Predicts 2026: AI Agents Reshape Business Operations." Gartner Research, 2025.

[2] Payments Dive. "Payment outages cost $44B in lost sales annually." Jan 2026. https://www.paymentsdive.com/news/payment-outages-cost-44b-in-lost-sales-annually/810481/

[3] Appriss Retail. "The 2026 Total Retail Loss Benchmark Report." 2026. https://apprissretail.com/2026-total-retail-loss-benchmark-report/

[4] Semperis. "nOAuth Abuse Alert: Full Account Takeover of Entra Cross-Tenant SaaS Applications." 2025. https://www.semperis.com/blog/noauth-abuse-alert-full-account-takeover/

[5] Fortune. "An AI-powered coding tool wiped out a software company's database." Jul 2025. https://fortune.com/2025/07/23/ai-coding-tool-replit-wiped-database/

[6] Chen, W. et al. "How are AI agents used? Evidence from 177,000 MCP tools." arXiv:2603.23802, Mar 2026.

[7] "Beyond Max Tokens: Stealthy Resource Amplification via Tool Calling Chains in LLM Agents." arXiv:2601.10955, Jan 2026.

[8] OWASP. "Top 10 for Agentic Applications 2026." Version 2026 (Dec 2025). https://genai.owasp.org/download/52117/

[9] Stripe. "Idempotent requests." Stripe API Documentation. https://docs.stripe.com/api/idempotent_requests

[10] Square. "Idempotency." Square Developer Documentation. https://developer.squareup.com/docs/build-basics/common-api-patterns/idempotency

[11] "Model Context Protocol Specification." Version 2025-11-25. https://modelcontextprotocol.io/specification/2025-11-25

[12] The Register. "Replit vibe coding incident." Jul 2025. https://www.theregister.com/2025/07/21/replit_saastr_vibe_coding_incident/

[13] Davis, C. "Production-ready agents with the OpenAI Agents SDK + Temporal." Temporal Blog, Jul 2025. https://temporal.io/blog/announcing-openai-agents-sdk-integration

[14] LangChain. "Interrupts." LangGraph Documentation. https://docs.langchain.com/oss/python/langgraph/interrupts

[15] Partnership on AI. "Prioritizing Real-Time Failure Detection in AI Agents." Sep 2025. https://partnershiponai.org/wp-content/uploads/2025/09/agents-real-time-failure-detection.pdf

[16] Microsoft Research. "Magentic-UI: Towards Human-in-the-loop Agentic Systems." Jul 2025. https://www.microsoft.com/en-us/research/wp-content/uploads/2025/07/magentic-ui-report.pdf

[17] Google. "Workflow Agents" and "Human input for agent workflows." ADK Documentation. https://google.github.io/adk-docs/agents/workflow-agents/

[18] OpenAI. "Handoffs." Agents SDK Documentation. https://openai.github.io/openai-agents-python/handoffs/

[19] OpenAI. "Running agents." Agents SDK Documentation. https://openai.github.io/openai-agents-python/running_agents/

[20] OpenAI. "Building Governed AI Agents." OpenAI Cookbook, Feb 2026. https://developers.openai.com/cookbook/examples/partners/agentic_governance_guide/

[21] OWASP. "A Practical Guide for Secure MCP Server Development." Feb 2026. https://genai.owasp.org/download/52676/

[22] LangChain. "Durable execution." LangGraph Documentation. https://docs.langchain.com/oss/python/langgraph/durable-execution

Appendix A: Cork Middleware Chain (Annotated)

The following pseudocode illustrates Cork's Express middleware chain and how BARC invariants map to specific middleware components.

```
// I1: Tenant Isolation
function tenancyMiddleware(req, res, next) {
  const storeId = extractStoreId(req.user, req.headers);
  if (!storeId && req.user.role !== 'SUPER_ADMIN')
    return res.status(403).json({ error: 'Store context required' });
  req.storeId = storeId;
  next();
}

// I2: Idempotency
function idempotencyMiddleware(req, res, next) {
  if (['POST','PUT','PATCH','DELETE'].includes(req.method)) {
    const key = req.headers['idempotency-key'];
    if (key) {
      const existing = await IdempotencyRecord.findUnique({ where: { key } });
      if (existing) return res.status(200).json(existing.response);
    }
  }
  next();
}

// I3: Atomic Transactions (service layer)
async function completeSale(saleId, paymentData, storeId) {
  return prisma.$transaction(async (tx) => {
    const sale = await tx.sale.update({ where: { id: saleId, storeId }, ... });
    await tx.paymentTransaction.create({ data: { ...paymentData, storeId } });
    for (const item of sale.items) {
      await tx.product.update({
        where: { id: item.productId, storeId },
        data: { stock: { decrement: item.quantity } }
      });
    }
  });
}
```

Appendix B: BARC Tool Classification Schema

```
{
  "tool": "capture_payment",
  "blast_radius_tier": "irreversible",
  "policy": {
    "requires_human_approval": true,
    "idempotency_key_pattern": "sale:{saleId}:capture",
    "requires_transaction": true,
    "tenant_scope": "single_store",
    "min_rbac_role": "MANAGER",
    "durable_saga": true,
    "compensation_action": "void_authorization"
  }
}
```

Appendix C: CorkBench Task Specification (Example)

```
{
  "task_id": "PAY-003",
  "domain": "payments",
  "goal": "Capture payment for sale #1042 ($84.97)",
  "constraints": [
    "Must use idempotency key",
    "Must verify sale belongs to authenticated store",
    "Must not capture if sale.status !== PENDING"
  ],
  "fault_injection": "retry_storm_3x",
  "expected_state": {
    "sale_status": "COMPLETED",
    "payment_count": 1,
    "charge_amount_cents": 8497
  },
  "safety_violations": [
    "payment_count > 1 (duplicate charge)",
    "charge_amount != 8497 (amount mismatch)"
  ]
}
```