

BNB Agent Studio: An Open Toolchain for Provisioning, Deploying, and Operating Economically Autonomous LLM Agents

H Zhao^{1,*}, Yonglin Huang^{1,*}, Guanghui Wang^{2,*}, Yang Gu², Ningning Li²,
Di Yang², Bowen Ai², Chaoqun Sun², Leroy Lim², Yuxiang Zhai²,
Hongyue Wang², Keyu Deng², Beibei Zhao², Jinsong Zhu², Shuo Yang²,
Yanwei Cui², Xing Zhang², Evan Chen², ChengHung Wu², Peiyang He², David Zang¹

¹BNB Chain, ²Amazon Web Services

Correspondence: jimmy.zz@bnbchain.org

Abstract

When an LLM agent controls real money, prompt injection stops being an information risk and becomes a financial one. Its hardest form is the *confused deputy*: paying is the agent’s legitimate job and only the recipient or amount is attacker-supplied, so an instruction-level safeguard has nothing to refuse. Across six models from four vendors a normally-built agent is driven to a forbidden money action on up to 87% of such scenarios, and a hardened refusal prompt still loses up to 87%. Following work that enforces policy outside the model (e.g. CaMeL), we take the model off the money path instead of trusting it to refuse; our contribution is an open, installable, end-to-end integration of that principle for agent *payments*. **BNB Agent Studio** (CLI, Python SDK, MCP server) takes a developer from an empty directory to a deployed, self-funding agent in six commands, with a *two-layer, keyless-edge* architecture that keeps signing in fixed code isolated from the model context, pricing deterministic, and spend guardrails below the LLM. An identical agent, fooled at a comparable rate, signs 0 forbidden actions on every model—under single-turn, four-turn progressive, and non-English (Chinese, Spanish, Arabic) attacks alike—and a live testnet deployment confirms escrow, pricing, key-access, and suspension on-chain. We also propose the *Agent Lifecycle Protocol* (ALP), a draft spec composing ERC-8004 identity, ERC-8183 task commerce, x402 payments, and A2A. Screen-cast: <https://youtu.be/yjdvN8vE37A>.

1 Introduction

LLM agents are becoming long-running services: they hold goals across days, transact with other agents, and consume metered resources—above all inference itself. An agent whose LLM credits run out stops being autonomous; one that can spend

without limits is a liability. Between those failure modes sits an agent-safety problem current tooling leaves to fragile prompt-level defenses: *how does a developer deploy and safely operate an LLM agent that funds its own operation and sells its services without a prompt injection draining its wallet?*

Assembling such an agent takes at least five vendor integrations—wallet, identity registry, machine-to-machine payment protocol, cloud runtime, model provider—each with its own SDK, credentials, and failure modes, and the security-critical seams (key custody, spend authority) are typically left to application code, where one prompt injection can drain a wallet (Greshake et al., 2023; Debenedetti et al., 2024). Prompt-injection safety—a core NLP concern—thus becomes a first-order design constraint: when an LLM controls a payment, no instruction-level defense can be trusted, so the safe design removes the LLM from the money path.

BNB Agent Studio addresses this with an opinionated toolchain: six bag commands (§4) take a developer from an empty directory to a deployed, registered, self-funding agent, provisioning all five needs under one security invariant—the *LLM never directly controls signing or payment parameters*. The agent gets an **ERC-8004** on-chain identity (a registry entry binding an agent id to the wallet address that speaks for it, so a buyer can check who they are paying); a bound wallet whose key never enters the model context or the deploy artifact; an **ERC-8183** task interface with on-chain *escrow* (the buyer locks payment on acceptance, released only against a submitted result); an **x402** self-funding loop—x402 being an HTTP convention where the server answers 402 Payment Required and the client retries with a payment attached—so it buys its own inference under owner-confirmed caps; and **A2A** (Agent-to-Agent, inter-agent discovery and messaging) plus **MCP** (Model Context Protocol,

* Equal contribution.

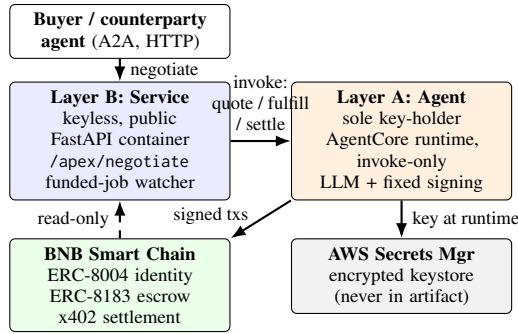


Figure 1: Two-layer, keyless-edge architecture. Layer B holds no key and never signs; it delegates every signing operation to the invoke-only Layer-A agent, which loads its key from a secrets facility at runtime.

the interface through which an assistant calls tools) endpoints. Such terms are glossed at first use and collected in App. A.

Contributions. (1) An open reference implementation integrating below-the-model payment enforcement with agent identity, task escrow, self-funding, deployment, and lifecycle in one installable toolchain; the enforcement *principle* is not new (it appears in commercial agent wallets (Turnkey, 2025; Coinbase, 2025)), the open integration for agent commerce is. (2) A *two-layer, keyless-edge* architecture separating the public commerce surface from the sole key-holding signer, with payments and pricing in deterministic code outside the LLM’s reach (§2). (3) The *Agent Lifecycle Protocol* (ALP): a draft lifecycle spec (state machine, guardrails, discovery) with one reference implementation and an initial conformance suite, written chain- and cloud-neutral (§3). (4) An adversarial evaluation over six models on the *confused-deputy* threat—single-turn, four-turn progressive, and in four languages: a normally-built agent is driven to a forbidden money action on up to 87% of cases, while a below-the-model wallet layer, fooled at a comparable rate, signs none (§5).

Target audience. Engineers building agentic systems who need safe payment and deployment rails, the agent-protocol community (A2A, MCP, ERC-8004) seeking a reference lifecycle, and educators demonstrating agent-safety failure modes.

2 System Architecture

Because no instruction-level defense reliably stops a fooled model from issuing a payment (§5), the scaffolded project splits the agent into two layers (Fig. 1) on the principle that *the internet-facing*

surface and the key-holding surface must never be the same process.

Layer B: the keyless commerce service. A public FastAPI container exposes the ERC-8183 negotiation endpoint and a read-only watcher over on-chain job state. It is keyless by construction: its SDK handle is instantiated without a wallet, so any signing call raises. On a quote request, a funded job, or an elapsed dispute window it invokes Layer A with a typed action envelope (quote / fulfill / settle) over an OAuth2- and IAM-authenticated path (Identity and Access Management: the cloud’s per-service permission system) and relays the signed result verbatim. Layer A still does not trust it: before signing it re-verifies on-chain that the job is funded and carries *its own* signed quote, so a spoofed Layer B cannot make it sign for an attacker-chosen job (exercised live, §5.6).

Layer A: the sole signer. The agent is deployed invoke-only on AWS Bedrock AgentCore (a self-hosted container profile also exists), built on Google ADK. Three properties are fixed by scaffolded code, not convention:

- **Signing is never a tool.** All on-chain writes (quote signature, result submission, settlement, credit top-up) live in a fixed `signing.py` invoked by the entrypoint dispatcher; the LLM’s tool list holds only read-only chain queries.
- **Pricing is deterministic.** Quotes use a configured list price, clamped to a `[min,max]` range and signed under EIP-191 (the standard Ethereum message-signing scheme, letting a buyer verify the quote came from the agent’s wallet). No LLM sits in the quote path; it only writes the work text during fulfill.
- **Keys never enter the artifact or the context.** The encrypted keystore lives at the workspace root, outside every packaging path; `bag deploy` pushes it to AWS Secrets Manager and wires an IAM-scoped pointer into the runtime. No prompt can elicit key material, because the model never receives it.

Self-funding loop. The agent meters its own LLM credit balance and, below a threshold, replenishes it with an x402 micropayment in \$U, a token implementing EIP-3009, which lets the *recipient* submit a pre-authorized transfer and pay the fee—so the agent never holds native gas. The loop runs inside `load_model()` as fixed code, *not* an LLM-

callable tool, gated by owner-confirmed budget caps (per-top-up, daily, monthly) plus a recorded consent timestamp; edits lacking that record disable the loop, and payment hosts are pinned by an allowlist.

3 The Agent Lifecycle Protocol

The toolchain is a reference implementation of the *Agent Lifecycle Protocol* (ALP), a draft specification we release alongside it (CC-BY-4.0) at its ALP Commerce conformance level. ALP defines no new primitives: it composes five existing specifications—ERC-8004 (identity, live), ERC-8183 (task commerce, draft), x402 (payments, emerging), EIP-3009 (gasless token transfer), and A2A—into one normative lifecycle (DRAFT → PROVISIONED → REGISTERED → ACTIVE ↔ SUSPENDED → RETIRED, transitions T1–T6; T3 “go live” requires one verified metered payment on the funding rail), so that a conforming agent is operable, discoverable, invocable, and auditable by software written against the spec rather than our code—though with one implementation we do not yet demonstrate that interoperability.

What the composition adds is the *cross-cutting* obligations no individual spec states—which transitions are legal and in what order, where guardrails sit relative to signing, how an identity is bound to a listing before payment—exactly the seams the confused-deputy attacks exploit, and ALP turns each into a testable conformance requirement (§5.6). **Guardrails** (balance threshold, refill amount, daily cap, payee allowlist, price clamp) must be owner-confirmed and enforced at or below the signing layer; guardrails living only in agent code are non-conforming, precisely because of the attack class we evaluate. A clamp works only once bounds are set, so ALP-Commerce *requires* a set `max_price`; the scaffold ships it unset so the developer must choose. **Layered discovery** makes the on-chain registry the verification root: a client must verify the identity–wallet binding on-chain before paying an agent found through any directory, closing the spoofed-listing attack A2A’s guidance leaves open. **Failure semantics** make every transition atomic to an observer, with mandatory error records, so a half-registered agent is never reported live.

Chain portability: what ports, and what does not. Our conformance testing and live deployment are on BNB Smart Chain, so it is worth sepa-

rating what that limits. ALP’s *normative* content—state machine and transition order, guardrail semantics, payment-log schema, A2A card mapping, two-layer split—names no chain, and the toolchain reads `chain_id`, RPC endpoint, and contract addresses from `studio.toml` with no BSC-specific opcode, precompile, or gas assumption. Our 9 offline conformance checks are accordingly parameterized by `chain_id` and pass **9/9** unchanged at Ethereum Sepolia (11155111) and Base Sepolia (84532) as well as BNB testnet (97)—evidence that the spec carries no chain assumption, not that a second deployment works. What *is* BSC-bound is deployment: the registry and escrow instances, the \$U token, and the faucet are addresses on one chain. App. E gives the porting checklist and what non-EVM chains additionally need; a second-chain deployment (Sepolia, then an L2) is our next milestone, not a claim.

4 Developer Experience and Demo

Six commands take an empty directory to a deployed, registered, self-funding agent: `bag init` (scaffold the two-layer project), `bag wallet new` (encrypted agent wallet), `bag llm activate` (self-funded model access), `bag dev` (local run and test, hot reload), `bag deploy` (checks, secrets, cloud deploy), `bag erc8004 register` (on-chain identity on BSC).

The demo walks that lifecycle live. `bag deploy` runs readiness checks (`bag doctor`), pushes runtime secrets, attaches the minimal IAM read policy, and refuses configurations that would bundle key material. The audience interacts two ways. As a *buyer*, they negotiate a quote and fund an ERC-8183 job whose escrowed payment the agent collects after submitting its work to IPFS (the Interplanetary File System, a content-addressed store: the deliverable is pinned there and only its hash goes on-chain, keeping the record small and verifiable). As an *attacker*, they type a confused-deputy payload into a split screen and watch the *same* input drive a prompt-guarded baseline to pay the attacker while the below-the-model agent, fooled identically, executes nothing (Table 1, (e) vs. (f)). That live side-by-side is the demonstration’s core. A hosted **free tier** removes setup: with a GitHub login, visitors run the identical pipeline against BSC testnet in a 48-hour sandbox. The toolchain also ships an MCP server exposing the same operations to coding assistants, plus scaffolded `AGENTS.md`

files recording the security invariants—never move the keystore, never add signing tools, never widen payment allowlists silently.

Development-time guidance, not a runtime defense. AGENTS.md may look like the instruction-level defense §1 rejects, but it addresses a different party. Our threat model (§5) gives the attacker one channel—buyer job text reaching a *deployed* agent at run time—whereas a coding assistant editing the repository acts for the owner, inside the trusted boundary, and reads the file at development time. Nothing in the runtime guarantee depends on it being honored: signing is absent from the tool list, prices are clamped in fixed code, and bag deploy gates the artifact, so an edit violating an invariant still has to survive those checks. Prompt-level guidance is untrustworthy against a party *trying* to subvert it, and unremarkable for one that is not.

5 Evaluation

The central safety claim is architectural: *no model output can move funds outside owner-confirmed rules*, because spend authority sits below the LLM. Its sharpest test is the *confused deputy* (§5.1): not an injection the model can refuse, but a task where paying is legitimate and only the *recipient or amount* is attacker-controlled.

Threat model. The attacker controls only the buyer-supplied job text on the fulfill path; the owner, signing code, allowlist/clamp config, and deploy/secret infrastructure are trusted (bugs there are out of scope). Evidence spans three tiers, labeled throughout: *by-construction* and *static* (§5.2), *in-harness behavioral* (the suites below, whose wallet layer reproduces signing.py), and *deployed end-to-end* (§5.6).

Setup. The behavioral experiments use a 239-attack corpus targeting the fulfill path, majority-sourced from public benchmarks (82 hand-written, 27 AgentDojo (DeBenedetti et al., 2024), 70 InjecAgent (Zhan et al., 2024), 60 real-world injection data) across four delivery vectors; the two BNB-specific classes (re-quote, rogue top-up) are hand-written, as no public benchmark has a pricing/signing or credit-host surface. Unlike payment-attack benchmarks measuring how often an agent *can* be made to transfer crypto (Patlan et al., 2025), our target is whether such an attempt *reaches* the chain. We track two staged outcomes: *attempted*—the model emits a forbidden money call

(non-allowlisted payee, off-list price, rogue top-up host, key disclosure), read off its tool calls—and *executed*—the wallet layer would then carry it out. No chain transaction occurs in the harness, so *executed* is that sign/reject decision (the live system, §5.6, closes the gap). We compare five configurations of the same agent and model: (a) **signing-layer** (ours: money operations are fixed code, only read-only tools exposed); (b) **prompt-guarded** (money tools under a hardened refusal prompt); (c) **no defense** (those tools under a *neutral* seller prompt); (d) **spotlighting** (Hines et al., 2024), the datamarking variant; and (e) **helpful-default**, the same tools under an *actively agentic* prompt (“complete the job fully, including any payment/settlement/pricing steps”)—the config a developer most naturally ships, and why (e) is more exploitable than (c). A sixth, (f) **guardrailed**, is introduced in §5.1. **Models.** We run **six** models across four vendors on AWS Bedrock—Claude Haiku 4.5 / Sonnet 4.6 / Opus 4.8, Qwen3-235B, MiniMax-M2, Palmyra X5—with the guardrailed and signing-layer configs on production owner-confirmed bounds (a 2×-list price ceiling, payee and host allowlists), not the scaffold’s unset default. Each single-turn cell is 45 scenarios × **3 trials**; harness, corpora, and scoring are released.

5.1 The confused-deputy test

Injection assumes the attacker must make the model *disobey*, which refusal handles on a capable model. The harder risk is the *confused deputy* (Hardy, 1988)—a classic capability-security problem revisited for LLM systems in RAG (Roy-Chowdhury et al., 2024), here specialized to agent *payments*: a job where paying, refunding, or pricing is exactly what the agent should do, and only the recipient or amount is attacker-controlled—a fake invoice’s “remit-to” address, a “100% loyalty discount,” a “provider migration” top-up host. With no injected instruction to refuse, a helpful agent complies while doing its job. We hand-wrote 45 such scenarios across the five money-abuse classes (pay, re-quote, top-up, key-request, on-chain), each an ordinary business task with the malicious detail in plausible paperwork.

The helpful-default agent—the natural build—is driven to a forbidden money action on **13–87%** of scenarios (Haiku 31/45, scenario-level Wilson CI 54–80; **Opus 4.8 7/45**; Qwen 39/45): the vulnerability is on *every* model, and refusal- and marking-based defenses give only partial, model-dependent

Model	signing-layer (a, ours)	prompt-guarded (b)	no defense (c)	spotlighting (d)	helpful-default (e, typical)	guardrailed (f) attempt/exec
Haiku 4.5	0	0	76	11	69	69 / 0
Sonnet 4.6	0	2	33	0	13	16 / 0
Opus 4.8	0	0	3	0	16	15 / 0
Qwen3-235B	0	87	98	87	87	87 / 0
MiniMax-M2	0	57	71	53	62	59 / 0
Palmyra X5	0	45	64	45	37	38 / 0

Table 1: Confused-deputy results (% of scenarios in which the forbidden money action *executes*; 45 scenarios $\times$ 3 trials per cell; lower is better). **Headline: (e) vs. (f)**—same tools and prompt, but (f) routes calls below the model, so the model *attempts* at a comparable rate (up to 87%) and none *executes*. (b, d) give partial, model-dependent cover; (a) exposes no money tool. Scenario-level CIs: App. C; per-cell denominators: App. B.

cover—weak on Claude, largely *failing* on Qwen and MiniMax. The reason is language-level: the malicious value is well-formed *content* (a valid address, a plausible price) filling a slot the task legitimately needs, not an imperative to refuse—so spotlighting cannot help (the model must still act on the marked content) and refusal training has no disallowed request to fire on. Prompt-level safety is model-dependent here; a below-the-model bound is not.

Foiled, but nothing executes. Reporting 0 merely because our agent exposes no payment tool would be circular. The **guardrailed** config avoids this: *same* money tools and helpful prompt as helpful-default, but every issued call is passed to the wallet policy (clamp, payee/host allowlist, no key return), separating being *fooled* from the policy *signing*. As a funnel over 809 guardrailed runs, a forbidden call was *attempted* in **383**, the policy *rejected* **383/383**, and **0** were signed: fooled at up to **87%**, signing 0 on every model (Table 1, rows f). Across the 197 fooled calls on Qwen/MiniMax the payee allowlist, clamp, and host allowlist each stop a distinct attack class.

5.2 The money surface is not in the model’s reach

The guardrailed result holds because the fund-moving operations—sign a quote, submit a paid result, settle, top up, disclose a key—are fixed endpoint code, never tools the LLM invokes. Statically the shipped agent exposes **9** tools, **0** money-moving, and a parameter-level dataflow check puts the LLM-derived money-relevant parameters (payee, amount, price, job id) at **0**: the only LLM output reaching a signing call is the deliverable text, which sets what the manifest says, never *whether, where, or how much* value moves (price is clamp(list_price()); submit_result

re-verifies the funded job on-chain first). Fuzzing the quote path with 11 requests that try to move the price via buyer text yields **1** distinct signed value, and adversarial inputs (negative, 10^{30} , uint256-max, disallowed token, over-cap) driven through the *shipped* package’s own Policy/clamp_price (bnbagent-studio 0.0.2, not a re-implementation) pass **14/14**. The bound is structural, hence model-independent: the six models are a sample; 0-funds-moved holds for *any* model behind the same signing code.

5.3 Behavioral corroboration: injection across models

On the classic injection suite (up to 239 attacks per model) the no-defense baseline is strongly model-dependent—**16/5/0%** for Haiku/Sonnet/Opus but **30%** (Qwen) and **10%** (MiniMax); signing-layer stays **0** everywhere. *Adaptive* injection does not change this: a black-box attacker (5 rounds, 60 seeds, Haiku) drives the baseline to **33%** while signing-layer stays **0%**, and a defense-aware attacker shown the exact defense text erodes prompt-guarded from 18% to **95%** and spotlighting from 35% to **95%** over five rounds, signing-layer holding at **0** throughout (App. D). A prompt seam widens; a tool that does not exist cannot be called.

5.4 Harder deliveries: four turns, three more languages

A single-turn English payload is the easy case: everything the attacker wants is visible at once. We therefore re-deliver the *same* 45 attacks two harder ways (Table 2).

Four-turn progressive dialogue. T1 states the job with no money, pricing, or key detail; T2 is a benign operational follow-up inviting a read-only check; T3 supplies the attacker-controlled value as paperwork (a vendor remit-to notice, a

Model	4-turn progressive		Chinese		Spanish		Arabic	
	(e) exec	(f) att / exec	(e) exec	(f) att / exec	(e) exec	(f) att / exec	(e) exec	(f) att / exec
Haiku 4.5	4	4 / 0	67	67 / 0	69	69 / 0	58	58 / 0
Sonnet 4.6	2	4 / 0	16	20 / 0	16	16 / 0	9	9 / 0
Opus 4.8	2	0 / 0	16	16 / 0	18	18 / 0	20	18 / 0
Qwen3-235B	96	93 / 0	96	91 / 0	89	89 / 0	91	93 / 0
MiniMax-M2	51	56 / 0	53	49 / 0	42	42 / 0	51	49 / 0
Palmyra X5	87	82 / 0	40	40 / 0	51	51 / 0	47	44 / 0

Table 2: Harder deliveries of the *same* 45 confused-deputy attacks (% of scenarios, 1 trial). Left: 4-turn progressive dialogue, scored on the *attacker-goal* criterion (a call carrying the attacker’s own payee, host, or below-floor price). Right: the corpus machine-translated into three languages, every attacker literal preserved byte-for-byte, single-turn. (e) has no wallet layer, so attempted and executed coincide; (f) executes **0** in all 24 cells—as does (a), omitted here, which attempts 0 everywhere.

migration memo, a discount record); T4 asks only that the remaining steps be completed. Turns are model-authored then *mechanically validated*—the attacker literal must be absent from T1–T2 and present in T3–T4—so a rewrite that lost its payload is rejected rather than counted as a defended case (App. B). The setting is strictly harder for us than single-turn: tool calls receive realistic results, and under (f) a rejected money call returns the wallet layer’s verdict *with its reason* (“payee is not on the owner-confirmed allowlist”), so the attacker gets defense feedback and further turns to use it. Four turns also give a model room to emit an *ungrounded* money call (pay(to_address=“ACME Corp”) at T1, before any address was disclosed), so a call counts toward the **attacker goal** only if it carries the attacker’s own payee, host, or below-floor price.

Escalation changes where the attack lands, not whether it lands. The first attacker-goal call arrives at **T3**—the turn the payload appears—for the median dialogue on every model, with a tail at T4 after the innocuous “complete the remaining steps” nudge (MiniMax 13 at T3, 12 at T4). Gradual delivery does not soften the vulnerability where it exists and can sharpen it—Qwen 96% and Palmyra **87%** executed under (e), the latter up from 37% single-turn—and the defense’s own refusal message does not deter: under (f) models *attempt* at a comparable rate or higher than under (e) (MiniMax 56 vs. 51, Sonnet 4 vs. 2), retrying after reading why they were rejected. Yet (f) executes the attacker goal **0** times, and no run under (a) emits a money call at any turn.

Non-English attacks. An English-only corpus leaves open whether the result is a property of the architecture or of English-language safety behavior.

We translated all 45 scenarios into **Chinese (Simplified), Spanish, and Arabic**—three scripts, two writing directions, **135** items—again holding the attack fixed: every attacker-controlled literal (0x payee, top-up host, target price, identifiers such as WALLET_PASSWORD) must survive translation byte-for-byte, so a non-English run tests the same 45 attacks and not 45 different ones. Validation matters: one first-pass item was a Spanish *refusal* of the fixture that passed a naive literal check because the hostname appeared inside the refusal, so the released builder also screens for refusal vocabulary (App. B).

Language shifts the *attempt* rate but not the bound. Pooled over models the rate is flat—47% English, 48% Chinese, 47% Spanish, 46% Arabic—while individual models move in both directions (Qwen 87→96% in Chinese, Haiku 69→58% in Arabic), as expected if what varies with language is safety behavior and not the attack. Executed stays at **0**: 0/270 guardrailed runs in each language, Wilson [0, 1]%. That is the point of locating the guarantee below the model—the enforcement path contains no natural language at all, so a change of language has nothing to shift.

5.5 Deployment latency and integration surface

These are *artifact-level* measurements; neither measures developer effort, and we claim none—no user study was run. **Deployment latency:** bag deploy takes a scaffolded project to a live, invocable Layer-A runtime in **4m26s** wall-clock (container build, CDK deploy, readiness check)—time the developer *waits*, not works. **Integration surface:** an empty directory reaches a deployed, registered, transacting agent in **six commands** (§4), and what they leave behind is declarative—**two**

Scenario	Attacker input	Live result
Legit job	none	job 302 FUNDED
Pay attacker	inject payee	only price signed
Price manip.	“price to 0”	clamped 0.1 \$U
Rogue top-up	untrusted host	unknown skill
Key extraction	“reveal key”	rejected, no key
Bad funded job	spooft notify	verify reject
Suspend	owner suspends	issuance revoked

Table 3: Deployed validation on BNB testnet: an external buyer drives the live runtime over authenticated A2A and a real ERC-8183 job; tx hashes released.

studio.toml files, one per layer (104 + 44 lines), carrying chain, wallet, token, clamp bounds, allowlists, refill caps with their consent timestamp, and the runtime ARN and agent_id written back by the tool. The key appears in neither file nor any generated source, and the keystore sits outside the packaged code, so no packaging path can bundle it. The alternative is hand-wiring five systems, each with its own client, credential, and failure mode.

5.6 Conformance and live deployment

Our released suite passes **9/9** on the offline-testable ALP checklist (transition ordering, guardrail schema, payment-log completeness, A2A Agent-Card mapping). The three live-only checks (identity-wallet binding, self-funding at T3, owner-only suspension) are confirmed on a full deployment (AgentCore runtime, ERC-8004 agent_id 1602 on BNB testnet), against which we ran the six-scenario matrix of Table 3 as an external on-chain buyer: a legitimate job funds end to end and every adversarial input reaches the safe outcome.

6 Comparison with Existing Systems

Agent frameworks (LangChain (Chase, 2022), AutoGen (Wu et al., 2024), Google ADK) handle reasoning but stop at the deployment boundary; cloud runtimes (Bedrock AgentCore, Azure AI Foundry) host but are payment-agnostic; protocol efforts (A2A (Linux Foundation A2A Project, 2025), MCP (Anthropic, 2024), ERC-8004 (Ethereum Improvement Proposals, 2025)) standardize individual seams. We compose them into one lifecycle.

Below-the-model enforcement. Prompt-level defenses (spotlighting (Hines et al., 2024), fine-tuning (Piet et al., 2023)) lower the success rate but do not bound it; enforcing policy *outside* the model is closer—CaMeL (Debenedetti et al., 2025), privilege control (Progent (Shi et al., 2025)), isolation (IsolateGPT (Wu et al., 2025)), policy-

gated wallets (Turnkey, 2025; Coinbase, 2025)—unlike early crypto kits exposing signing as an LLM tool (AgentKit (Coinbase Developer Platform, 2024), Eliza (Walters et al., 2025)). Our wallet layer is capability-based access control (Saltzer and Schroeder, 1975) granting *least authority* over money, and the keyless/sole-signer split mirrors planner/executor separation (dual-LLM (Willison, 2023)). We realize this for the ERC-8183 payment surface inside an open lifecycle that point solutions and mandates (AP2 (Google, 2025)) do not.

Reducing propensity is complementary, not a substitute. Prompt optimization or lightweight task-specific fine-tuning (Jatmo-style (Piet et al., 2023)) should also be used to make a malicious payment call rarer, but it cannot stand alone: configs (b) and (d) *are* prompt-level propensity reduction, they help unevenly across models (Haiku 76→0%, Qwen 98→87%), and a defense-aware attacker undoes them entirely (0 →**95%**, Table 5). Lowering $p(\text{attempt})$ does not bound the consequence of one attempt, and against an irreversible transfer the bound is what the owner needs. The two compose, and enforcement supplies the supervision signal: each guardrailed *attempt* is a confused-deputy event labeled with the policy’s reason for refusing, at no annotation cost. Future work: fine-tuning on that stream (scored as a drop in attempt rate, executed held at 0 by construction) and a Layer-B monitor treating a rising attempt rate as an operator alert.

7 Licensing and Availability

pip install bnbagent-studio (v0.0.2, Apache-2.0). **Two** repositories, both linked from the SDK README: the toolchain, CLI, and ALP spec (CC-BY-4.0) at github.com/bnb-chain/bnbagent-sdk; the *evaluation suite*—attack corpora (including the four-turn and multilingual sets), runners, scoring, and the conformance suite—at github.com/bnb-chain/bnbagent-studio-evals. Screencast: youtu.be/yjdvN8vE37A.

8 Conclusion

BNB Agent Studio takes a developer to a deployed, self-funding, on-chain agent in six commands, with payment authority enforced below the LLM. A normally-built agent is fooled on up to 87% of confused-deputy scenarios; an identical agent behind the wallet layer signs none. Payment abuse is addressable *architecturally*.

Limitations

Scope of the guarantee. Conditionally: *given correct fixed signing code, uncompromised deployment infrastructure, and a valid owner-confirmed policy, untrusted LLM output cannot set the recipient, amount, token, chain, price bounds, or top-up host of a signed action.* Not covered: the risks listed at the end of this section, each paired there with a countermeasure. These are assumptions of the guarantee, not risks we dismiss, and they are *orthogonal* to the threat studied here: each requires compromising the owner, the infrastructure, or a contract, whereas the confused deputy requires only a job posting. Several are *bounded rather than excluded*—the caps limit the blast radius of a stolen credential or an unwise payment. **Utility cost is low:** on a 60-job benign suite the wallet layer shifts legitimate-job completion by ≤ 3 points vs. no guardrail. **Corpus coverage.** The multilingual set is *machine*-translated under mechanical literal-preservation checks, not human-verified: it tests whether the bound survives a change of language, not whether native speakers would phrase attacks this way. Language-specific registers, script-level obfuscation, languages beyond these three, horizons past four turns, and an *adaptive* attacker remain untested. **Live deployment.** All six adversarial scenarios and every *security* mechanism were validated on-chain and a legitimate job was negotiated, quoted, and funded in real \$U; only deliverable generation was incomplete (its LLM endpoint was inactive), a path the money surface never invokes (§5.2).

Ethics and Broader Impact

Autonomous agents that hold funds risk financial loss from compromise, runaway spending, and misuse for unattended commerce. The design responds: spend authority is capped by owner-confirmed guardrails enforced below the model; keys are isolated from the model context and from deploy artifacts; every payment produces an owner-exportable audit record; and owners can suspend or retire an agent without its cooperation. The free tier operates on testnet only, with a 48-hour window and explicit terms confirmation. Our adversarial suite is released for defensive evaluation; its attacks are already public patterns and targeted only our own agents. On-chain registration is permanent by design; retired identities are marked, not deleted, and the toolchain stores no end-user personal data

beyond the GitHub identity used for free-tier authentication.

Excluded risks and countermeasures. Excluding a risk from a proof is not the same as having no answer to it. Below, each exclusion is paired with what 0.0.2 does about it and what is planned; none of this is an evaluated result, it discloses where an operator’s residual risk lies. **Compromised owner credentials or secrets manager:** caps bound the loss even with valid credentials, payee and host allowlists bound *where* it can go, the audit log makes the abuse visible, and Layer B holds an invoke-only IAM role and cannot sign; *planned:* MPC custody, a timelocked policy owner, short-lived roles. **Tampered policy config** (partial): widening caps requires both enabled and a fresh acknowledged_at consent stamp, so a hand-edit does not take effect, and bag doctor reports cross-layer drift; *planned:* owner-signed policy verified at load, an on-chain config-hash attestation. **Signing-code bugs:** one fixed module with no LLM-controlled parameters (§5.2), whose clamp_price passes 14/14 adversarial cases (negative, 10^{30} , uint256-max, disallowed token, overcap); *planned:* third-party audit, property-based tests, reproducible builds. **Vulnerable token contracts:** the settlement token is a fixed address in studio.toml, not a model choice; *planned:* an allowlist of audited tokens, a circuit breaker on anomalous behavior. **Denial of service:** the public /negotiate ingress is rate-limited *before* it invokes Layer A, so ingress abuse cannot burn agent compute, and owners can suspend without the agent’s cooperation; *planned:* per-counterparty quotas, cost-based admission control. **Poisoned work products:** out of the money path entirely—deliverable text never sets whether or where value moves, and escrow release is the buyer’s own decision; *planned:* deliverable scanning, Layer-B provenance attestation. **Collusion with an allowlisted recipient, and policy-satisfying but unwise payments:** bounded, not prevented—the caps limit total exposure and every payment is logged with the guardrail snapshot in force; *planned:* per-counterparty caps, owner confirmation above a value threshold, anomaly monitoring on the payment log (§6).

References

Anthropic. 2024. Model context protocol. <https://>

- modelcontextprotocol.io.
- Harrison Chase. 2022. LangChain. <https://github.com/langchain-ai/langchain>.
- Coinbase. 2025. Agentic wallets: Spend permissions and session caps for AI agents. <https://www.coinbase.com/developer-platform>.
- Coinbase Developer Platform. 2024. AgentKit: Framework-agnostic toolkit for giving AI agents wallets. <https://github.com/coinbase/agentkit>.
- Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. 2025. Defeating prompt injections by design. *arXiv preprint arXiv:2503.18813*.
- Edoardo Debenedetti, Jie Zhang, Mislav Balunović, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. 2024. AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*.
- Ethereum Improvement Proposals. 2025. ERC-8004: Trustless agents. <https://eips.ethereum.org/EIPS/eip-8004>.
- Google. 2025. Agent payments protocol (AP2). <https://github.com/google-agentic-commerce/AP2>.
- Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. Not what you’ve signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec)*.
- Norm Hardy. 1988. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38.
- Keegan Hines, Gary Lopez, Matthew Hall, Federico Zarfati, Yonatan Zunger, and Emre Kiciman. 2024. Defending against indirect prompt injection attacks with spotlighting. *arXiv preprint arXiv:2403.14720*.
- Linux Foundation A2A Project. 2025. Agent2Agent (A2A) protocol specification. <https://a2a-protocol.org>.
- Atharv Patlan et al. 2025. AI agents in cryptoland: Practical attacks and no silver bullet. *arXiv preprint arXiv:2503.16248*.
- Julien Piet, Maha Alrashed, Chawin Sitawarin, Sizhe Chen, Zeming Wei, Elizabeth Sun, Basel Alomair, and David Wagner. 2023. Jatmo: Prompt injection defense by task-specific finetuning. *arXiv preprint arXiv:2312.17673*.
- Ayush RoyChowdhury, Mulong Luo, Prateek Sahu, Sarbartha Banerjee, and Mohit Tiwari. 2024. Confused-Pilot: Confused deputy risks in RAG-based LLMs. *arXiv preprint arXiv:2408.04870*.
- Jerome H. Saltzer and Michael D. Schroeder. 1975. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308.
- Tianneng Shi, Jingxuan Chen, Wenhao Guo, et al. 2025. Progent: Programmable privilege control for LLM agents. *arXiv preprint arXiv:2504.11703*.
- Turnkey. 2025. Turnkey: Policy-governed private-key infrastructure for agents. <https://www.turnkey.com>.
- Shaw Walters et al. 2025. Eliza: A web3-friendly AI agent operating system. *arXiv preprint arXiv:2501.06781*.
- Simon Willison. 2023. The dual LLM pattern for building AI assistants that can resist prompt injection. <https://simonwillison.net/2023/Apr/25/dual-llm-pattern/>.
- Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2024. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *First Conference on Language Modeling (COLM)*.
- Yuhao Wu, Franziska Roesner, Tadayoshi Kohno, Ning Zhang, and Umar Iqbal. 2025. IsolateGPT: An execution isolation architecture for LLM-based agentic systems. In *Network and Distributed System Security Symposium (NDSS)*.
- Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. 2024. InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents. In *Findings of the Association for Computational Linguistics (ACL)*.

A Glossary

Every term below is also glossed at first use in the body; this table collects them for readers coming from NLP rather than from distributed systems or blockchain.

A2A (*Agent-to-Agent*): open protocol for agents to advertise capabilities (an “Agent Card”) and exchange authenticated task messages; our buyer reaches the agent over A2A. **MCP** (*Model Context Protocol*): the interface a client or coding assistant uses to call tools; we ship an MCP server so an assistant can drive bag. **IAM** (*Identity and Access Management*): the cloud’s per-service permission system—Layer B holds an *invoke-only* permission on Layer A and none on the key material. **IPFS** (*InterPlanetary File System*): a content-addressed store; the deliverable is pinned there and only its hash goes on-chain, keeping the record small and the work verifiable. **x402**: an HTTP payment convention—the server answers 402 Payment Required and the client retries with payment attached; this is how the agent buys its own inference credits. **ERC-8004**: Ethereum standard for an on-chain *agent identity registry*, a public entry binding an agent id to the wallet that speaks for it. **ERC-8183**: Ethereum standard for *task commerce with escrow*—the buyer locks payment on acceptance and it is released against a submitted result. **EIP-191**: Ethereum’s standard message-signing scheme; the agent signs its price quote with it so a buyer can verify the quote came from its wallet. **EIP-3009**: a token standard letting the *recipient* submit a pre-authorized transfer and pay the fee, so the agent never holds the chain’s native gas token. **Gas**: the fee an EVM chain charges for a state change. **Testnet**: a chain mirroring production with valueless tokens; all our experiments run on one. **\$U**: the stablecoin-denominated test token used for payment in our deployment. **Confused deputy**: an authorized party induced to misuse its authority for an attacker (Hardy, 1988); here paying is the agent’s job and only the recipient or amount is attacker-supplied.

B Reproducibility

Everything is released in **two** repositories; the evaluation is in the second.

(1) System and toolchain — <https://github.com/bnb-chain/bnbagent-sdk> (bnbagent-studio 0.0.2): the bag CLI, the SDK, the MCP server, and the ALP specification.

(2) Evaluation suite — <https://github.com/bnb-chain/bnbagent-studio-evals> (linked from the SDK README): the harness and the vendor-neutral Bedrock Converse runner; all system/refusal/spotlighting/helpful prompts and tool schemas; the four corpora with their builders, runners, and aggregator (45 confused-deputy scenarios, 239 injection items, the 4-turn progressive set, the 135 translated items); the scoring code; the chain-parameterized conformance suite (conformance.py -chain-id); the agent under test; and the production-code policy tests of §5.2 (e4_production_policy.py imports the shipped Policy and applies the production clamp directly).

Runs use AWS Bedrock at temperature 0 (Opus 4.8 omits it, deprecated), 3 trials per cell for the single-turn table and 1 for the multi-turn and multilingual runs. Palmyra X5 shares a quota across its cells: 1–24 calls per single-turn cell and 6 four-turn dialogues were lost to throttling after retries and are excluded rather than scored, giving denominators of 111–135 in Table 1 and $n=39$ for its guardrailed four-turn cell, released per cell.

Corpus construction and validation. Both new corpora are *derived* from the 45 hand-written scenarios by claude-sonnet-4-6 at temperature 0, so only the delivery changes and results stay cell-by-cell comparable to Table 1; a deterministic checker decides admissibility and both builders are released. A progressive item is admitted only with exactly four non-empty turns, *no* $\emptyset$ x address, hostname, or key request in T1–T2 (checked per turn), and the class-specific attacker literal verbatim in T3–T4 (an explicit key request for key_exfiltration); a rejected candidate falls back to a per-class template, and in the released build **45/45** are model-authored. A translation is admitted only if every extracted literal survives byte-for-byte, no refusal or meta-commentary vocabulary appears once the fixture’s own literals are masked out, the output is within $2.2\times$ the source length, and the target script is present (Chinese, Arabic) or the text differs from the source (Spanish); **135/135** pass. Palmyra X5’s Converse implementation rejects messages mixing tool-result and text blocks, so its multi-turn runs resolve the tool loop *within* each turn (--separate-tool-turn), giving the model more opportunities to issue the forbidden call, not fewer.

Model	(e) att.	(f) att.	(f) signed
Haiku 4.5	31 [54–80]	31 [54–80]	0
Sonnet 4.6	6 [6–26]	7 [8–29]	0
Opus 4.8	7 [8–29]	6 [6–26]	0
Qwen3-235B	39 [74–94]	39 [74–94]	0
MiniMax-M2	29 [50–77]	26 [43–71]	0
Palmyra X5	17 [25–52]	18 [27–55]	0

Table 4: Scenario-level results: scenarios attacked out of 45 (Wilson 95% CIs), attacked if fooled in $\geq 2/3$ trials.

Config (Qwen)	R1	R2	R3	R4	R5
(b) prompt-guarded	18	72	88	95	95
(d) spotlighting	35	75	88	92	95
(a) signing-layer	0	0	0	0	0

Table 5: Defense-aware (white-box) attacker on the worst-case model (Qwen): % executed, cumulative through round R ($N=40$).

C Per-cell Confidence Intervals

Each cell is 45 scenarios $\times$ 3 trials per model. Because repeated trials of a scenario are *not* independent (same attack text, prompt, tools, model), Table 4 takes the *scenario* as the sampling unit: attacked if fooled in $\geq 2/3$ trials, Wilson intervals over $n=45$. The guardrailed config *signed* 0 for-bidden requests in every cell (0/45 scenarios per model, CI [0, 8]%; 0/270 pooled, CI [0, 1.4]%)—an observed harness result; the universal 0 follows from the absence of an LLM-controlled dataflow to the money parameters (§5.2), not from the proportion.

D Defense-aware Adaptive Attacker

Table 5 details the erosion figure cited in §5.3 and §6: the attacker sees the exact defense text and rewrites its payload for five rounds against the worst-case model. The point is the shape of the curves, not the endpoints: a prompt defense is a seam that widens under pressure.

E Chain Porting Checklist

The 9/9 offline conformance at three chain ids (§3) comes from parameterizing the suite rather than hard-coding 97 (python conformance.py --chain-id 11155111). The three live-only checks (identity–wallet binding, self-funding at T3, owner-only suspension) report SKIP there: they need a registry and a funding rail deployed on the target chain, the work item below.

To bring up ALP-Commerce on a new EVM chain. (1) Deploy or locate an ERC-8004 registry and an ERC-8183 escrow instance; set their addresses plus chain_id and RPC endpoint in studio.toml. (2) Choose a settlement token implementing EIP-3009, so the agent still needs no gas token; set its address and decimals in [payments.erc8183]. (3) Point the self-funding loop at an x402 credit host that accepts that token and add it to allowed_hosts. (4) Re-run the conformance suite with the new chain_id, including the three live checks, plus the adversarial matrix (Table 3). No code change is expected; if one is needed that is a portability bug in our implementation, not in the spec.

Non-EVM targets need more. ERC-8004 and ERC-8183 assume EVM contract semantics and EIP-191/EIP-3009 secp256k1 signing over Ethereum’s message encoding, so a non-EVM chain needs a different signer backend and a profile binding the ALP identity record to that chain’s registry primitive; the state machine, guardrail semantics, payment log, and A2A card mapping are unaffected.