HIVM

HZB Network

面向Web3可信智能执行的
分层虚拟机架构与安全机制研究

HIVM: Research on a Layered Virtual Machine Architecture
and Security Framework for Trusted Intelligent Execution in Web3

HZB Network 技术研究组

2026 年 8 月

目 录

摘要

大型语言模型与智能体正从“生成内容”走向“调用工具并改变外部状态”，但只有进入真实任务、受控执行与可验证协作，才能形成持续生产力。HZB Network 将自身定义为 Web3 AI 基础设施与全球共建网络，核心主张是“让智能可调用，让建设可验证，让价值可结算，让创业可全球协同”。在 Web3 场景中，模型输出可能触发合约部署、交易构造、资产授权、治理投票或跨链调用，因此传统聊天式 AI 架构面临三类矛盾：概率性推理与确定授权、高性能链下计算与共享信任，以及区块链可证明状态与承诺，却无法自动证明模型结论、输入或业务决策的正确性。

HIVM 不是复制 Ethereum Virtual Machine（EVM）的链上字节码环境，也不以“所有 AI 计算上链”为目标，而是位于模型、数据、工具、钱包与多链网络之间的运行时抽象。它将自然语言或 API 请求编译为包含身份、权限、预算、数据分类、风险等级和成功条件的结构化任务，通过 Task Compiler、Model Router、Tool Runtime、Policy Engine、Verification Router、Chain Adapter 与 Receipt Engine 形成受控执行闭环，并根据任务复杂度、敏感度、价值风险和可验证性，在链下执行与链上可信协同之间动态分配工作负载。

本文给出任务元组、能力元组、策略判定、风险评分、验证等级、状态迁移和证据承诺的形式化描述，设计 READ、ANALYZE、GENERATE、SIMULATE、VALIDATE、AUTHORIZE、EXECUTE 与 ATTEST 八类指令，并提出 HIVM Execution Receipt，作为连接模型版本、工具调用、数据快照、策略结果、授权依据、执行事件和验证证据的统一审计对象。系统以零信任、最小权限、能力白名单、沙箱、网络出口控制、交易模拟、显式授权、限权钱包、签名域隔离、密钥分区和应急暂停构成纵深防御，并针对提示注入、检索污染、工具越权、供应链替换、RPC 操纵、重放、回执伪造与跨租户泄漏建立威胁模型。

HIVM、PoB 与开放网络目前均处于规划或开发阶段，本文不将设计目标表述为生产实绩。四组 HIVM 合成模拟仅比较策略门禁、双路径、声明级验证和回执完整性；PoB 则作为未来协议验证设计。分析与示例表明，可信智能执行的关键是将概率性能力编译为受约束任务，把外部副作用绑定到策略与授权，再将任务回执聚合为可复核、可争议、可结算的建设证明。HIVM 为 Web3 LLM、Developer API/SDK、Smart Contract Tools 与 AI Agent 提供统一运行时；Proof of Build 与 Venture Cell 则将产品能力连接至全球建设网络，构成 HZB 第二增长引擎。

关键词： HZB Network；HIVM；Web3 AI；人工智能虚拟机；智能体；可信执行；Proof of Build；建设证明；零信任；执行回执；Venture Cell；多链适配

Abstract

Large language models and AI agents are evolving from content generators into systems capable of invoking tools and producing external side effects. In Web3 environments, an AI output may lead to contract deployment, transaction construction, asset authorization, governance actions, or cross-chain calls. This creates an architectural mismatch between probabilistic reasoning and the deterministic requirements of authorization, state transition, auditability, and recovery. It also creates a placement problem: high-throughput inference, retrieval, and code analysis are better suited to off-chain compute, while identity, authorization, commitment, and settlement benefit from on-chain verification. Moreover, blockchain integrity does not by itself establish factual truth, model correctness, or the legitimacy of a business decision.

This paper treats HZB Network as a Web3 AI infrastructure and global co-building network, and proposes the HZB Intelligence Virtual Machine (HIVM) as its controlled execution, verification, and scheduling runtime. HIVM is neither a copy of the EVM nor a claim that all AI computation should be placed on-chain. It compiles user, API, agent, or venture-building intent into structured tasks carrying identity, permissions, budget, data classification, risk level, evidence requirements, and success conditions. The architecture follows the HZB principle that intelligence is formed off-chain while trust and value closure are established on-chain.

The paper formalizes task and capability tuples, policy predicates, risk scoring, verification levels, state transitions, execution receipts, build tasks, Proof of Build validity, and settlement gates. HIVM receipts bind model and tool versions, data snapshots, policy decisions, authorization evidence, execution events, and verification methods; PoB then aggregates eligible receipts and human deliverables into milestone-level contribution proofs. A defense-in-depth architecture addresses both agent execution attacks and network-specific threats such as duplicate contribution claims, verifier collusion, Sybil identities, cross-task replay, and premature settlement. Because the architecture remains planned or under development, all evaluations are reproducible prototype simulations and acceptance targets rather than production measurements.

Keywords: HZB Network; HIVM; Web3 AI; artificial intelligence virtual machine; AI agent; trustworthy execution; Proof of Build; zero trust; execution receipt; venture collaboration

1 引言

1.1 背景与矛盾

通用大模型已经具备语言理解、代码生成、规划分解和工具调用能力，但“能够提出建议”与“能够安全执行动作”之间存在结构性鸿沟。普通知识问答的失败多表现为答案错误；Web3 智能体的同类错误却可能转化为错误授权、不可逆资产转移、治理状态变化或跨链连锁损失。模型输出、检索文本和工具说明由此不再只是内容，而成为可能影响控制流的非可信输入。生成能力越强，越需要把“建议权”与“执行权”分开。

区块链提供确定性状态机、可编程授权、公开事件和可复核结算，但共识复制、存储费用和确定性约束使其不适合承载长上下文推理、大规模向量检索、多轮代码分析或频繁模型路由。EVM 的核心是对确定性字节码和链上状态的一致执行[1-3]；模型推理则可能依赖随机采样、动态知识和外部服务。把全部智能计算强行上链会放大成本、延迟与隐私问题；把全部过程留在传统后端，又会使身份、授权、版本与证据退化为平台自证。

HZB 采用“链下形成智能，链上建立信任与价值闭环”的混合原则：检索、推理、代码分析、交易构造与模拟主要位于链下；身份锚点、授权状态、承诺、关键事件、结算、信誉与争议引用按需进入链上。HIVM 位于二者之间，把自然语言或 API 意图转化为可检查任务，把模型与工具置于策略约束中，并把高风险候选动作提升为需要验证、显式授权和最终性观察的受控动作。这里的“虚拟机”指任务语义、能力接口、状态迁移和回执协议构成的运行时抽象，而不是新的操作码链或 EVM 复制品。

上述运行时解决的是“智能怎样被安全调用”，而 HZB 的第二个问题是“跨地域建设怎样形成可验证贡献”。第一增长引擎由 Web3 LLM、Developer API/SDK、Smart Contract Tools 与 AI Agent 构成可调用、可计量、可验收的专业基础设施；第二增长引擎由 Proof of Build 与 Venture Cell 把真实需求转化为任务、证据、信誉和价值结算。二者通过 HIVM Execution Receipt 连接：产品调用产生任务级证据，建设协议把多个任务级证据聚合为里程碑级建设证明。

1.2 问题定义与研究问题

给定主体、意图、数据、模型、工具、多链状态和组织策略，可信运行时必须连续回答：目标如何无歧义表示；本次任务可以调用什么；模型输出何时仅是建议；何种证据足以进入授权；

谁可对哪一项确定调用签名；外部状态发生后如何区分已提交、已确认、部分成功与已补偿；以及审计者能验证什么、不能验证什么。

据此，本文提出五个研究问题。RQ1：如何将自然语言目标编译为可授权、可计量且可追踪的结构化任务？RQ2：如何依据资产、隐私、不可逆性和证据条件选择链下/链上职责及验证强度，而不让高风险任务因成本偏好退化？RQ3：如何将零信任和能力安全落实为逐动作、逐参数、可撤销的外部副作用控制？RQ4：如何构造执行回执，既证明内容完整性、来源与过程绑定，又不把这些性质误写成输入真实、模型结论正确或业务决定合理？RQ5：如何把多个任务回执、人类交付物与独立复核绑定为 PoB 建设证明，并在申诉窗口关闭前阻止重复申领、串谋验证和提前结算？

1.3 研究贡献与证据口径

本文贡献包括五方面：第一，提出与 HZB 定位一致的五层架构和双增长引擎，把 HIVM 界定为基础设施内核而非项目全部；第二，以八类指令、双路径、状态机与幂等控制外部效果；第三，给出任务、能力、回执、PoB 与结算门的形式化模型；第四，给出七份接口级参考代码；第五，报告四组 HIVM 合成结果，并单列一组不含伪造数据的 PoB 前瞻验证方案。

本文严格区分“架构提出”“参考实现”“模拟观察”“未来验收”与“生产事实”。没有公开部署、审计或第三方复测证据时，不使用“已实现绝对安全”“已达生产吞吐”或“已形成主网”等表述。ChainGPT AIVM 仅依据其公开概念页和白皮书作技术对标[4-5]，不暗示与 HZB Network 存在合作、授权、兼容认证或技术继承关系。

1.4 文章结构

第 2–7 章依次建立理论模型、五层架构、运行时、参考代码与安全体系；第 8 章报告四组 HIVM 合成结果及一组 PoB 前瞻验证设计；第 9 章说明 HZB、HHA、Venture Cell、PoB 与 HIVM 的边界及路线；第 10 章总结限制和结论。

2 相关工作

2.1 从确定性虚拟机到智能执行运行时

虚拟机的共同价值在于以稳定抽象隔离程序语义和底层资源。EVM 通过栈、内存、存储、Gas 与全局状态转换，使节点对同一交易达成一致执行结果[1-3]。HIVM 借用“统一指令、资源计量、状态和回执”的思想，但不复用 EVM 的共识复制目标。EVM 执行对象是确定性字节码；HIVM 管理的是可能含概率推理的任务图，其安全判断依赖冻结的结构化工件，而不依赖逐字复现模型输出。二者是调用关系：HIVM 的 Chain Adapter 最终可调用 EVM 或其他既有执行层。

ChainGPT 公开资料将 AIVM 描述为面向 AI 模型、算力和数据的专用区块链基础设施，并在白皮书中讨论链上/链下混合执行及证明机制[4-5]。HIVM 与之共同关心 AI 计算和区块链验证的衔接，但边界不同：AIVM 的公开定位以 AI 原生网络与资源模块为中心；HIVM 不提出新的共识、专属代币或计算市场，而聚焦异构系统之间的任务编译、能力门禁、验证路由和回执。此处对标只说明公开技术方向的异同。

表 1　EVM、公开 AIVM 思路与 HIVM 的研究边界

维度	EVM	ChainGPT AIVM（公开资料）	HIVM（本文规划架构）
核心对象	确定性字节码与链状态	AI 原生链、模型/算力/数据模块	任务、能力、证据、工具、钱包与多链动作
执行位置	共识节点复制执行	公开资料描述混合路径	链下智能计算、链上可信协同
“VM”语义	操作码执行机	AI 专用网络基础设施	受控任务运行时与状态机
主要保证	给定输入与状态下的一致转换	以项目公开白皮书主张为准	最小权限、过程绑定、可审计回执的设计目标
本文关系	被 Chain Adapter 调用	仅作公开技术对标	研究对象，不声称已部署

2.2 RAG、推理—行动与外部副作用

RAG 把参数化生成模型与外部非参数记忆结合，使知识更新和来源引用更可控[8]。然而，索引过期、恶意语料、召回偏差和间接提示注入仍可能影响输出。HIVM 因而将检索视为带来源、许可、时间、片段哈希与索引版本的受控数据能力，而不是正确性证明。ReAct 将推理与行动交错，使模型能够使用工具观察调整计划[9]；其优势也扩大了攻击面，因为工具返回可以改变

后续计划。HIVM 在“模型提出动作”和“系统执行动作”之间插入模式校验、能力检查、模拟、验证和授权。

2.3 零信任、能力安全与风险治理

NIST AI RMF 及其生成式 AI 配置文件强调跨生命周期治理、度量和持续管理，而非以单次准确率替代系统风险[6-7]。零信任架构要求不因网络位置、既有会话或组件名称获得隐式信任，每次访问都针对主体、资源与上下文重新认证和授权[10]。Saltzer 与 Schroeder 提出的最小权限、完全仲裁、失效安全默认和权限分离原则[11]，以及能力系统关于“不可伪造引用承载有限对象权限”的工作[12-13]，共同构成 HIVM 的控制基础。能力令牌负责表达“最多能做什么”，策略点负责判断“这一次是否可做”，二者不能由模型置信度替代。

2.4 证明机制及其边界

哈希承诺可检测承诺后内容变化；数字签名可绑定签名主体和字节串；链上事件可证明规范链中某状态转换发生；TEE 远程证明可在硬件、TCB 和验证策略假设下说明某软件度量运行于特定环境[17-18]；zkML 可证明被电路编码的模型计算关系[19]；独立复算和人工复核则可覆盖部分难形式化语义。它们回答的是不同问题，不能拼接成“AI 绝对正确”。HIVM 将可验证性拆为完整性、来源、执行环境、规则执行、结果复核和业务确认六层，回执必须记录采用了哪一层以及未覆盖什么。

3 理论基础与形式化模型

3.1 受控 Agent 与安全动作集

将工具增强 Agent 视为部分可观测环境中的策略：时刻 t 的信念状态为 b_t，执行环境为 e_t，任务为 T，候选动作全集为 $\mathcal{A}$。模型只负责提出候选动作，参考监控器依据能力和策略得到安全可行集：

$$\mathcal{A}_t^{\text{safe}}(T) = \{a \in \mathcal{A} | \text{Decide}(T, a, e_t) = \text{Permit}\}, \qquad a_t \sim \pi_\theta(a|b_t, T)|_{\mathcal{A}_t^{\text{safe}}}. \tag{1}$$

式（1）表达“模型提出、系统裁决”：只有策略明确返回 Permit 的动作才进入可执行集合；Escalate 仅能触发 WAITING_AUTH，不能因其不等于 Deny 而被视作可执行。若安全集为空，运行时应拒绝、降级为只读分析或请求审批，不能允许模型自行扩权。它把模型错误的影响上界从“任意外部动作”缩小为“策略明示允许的动作子集”，但前提是参考监控器、能力存储和签名区未同时失陷。

RAG 的基本边缘化形式为：

$$p(y|q) = \sum_{z \in \text{TopK}(q)} p_\eta(z|q)\, p_\theta(y|q, z). \tag{2}$$

答案同时依赖检索器和生成器，所以 HIVM 不只记录最终文本，还记录检索快照、片段摘要与生成配置。可追踪性提高了复核能力，却不能消除召回遗漏或同源错误。

3.2 任务、能力与策略

一次入口请求被编译为任务元组：

$$T = \langle id, v_T, sub, g, I, O, Ctx, Con, B, D, \rho, \nu \rangle. \tag{3}$$

其中 id, v_T 为任务标识与冻结版本，sub 为主体，g 为规范化目标，I, O 为输入引用与输出模式，Ctx 为显式上下文，Con 包含合规、隐私、允许链和确定性约束；$B = (b_c, b_g, b_m)$ 分别约束货币成本、Gas 与模型调用额度，D 是截止时间，ρ 是风险标签，ν 是防重放随机数。任何影响计划、输入、权限或验证配置的变化都产生新 v_T。

主体的最小授权表示为能力元组：

$$c = \langle cid, ver, rev, iss, sub, obj, ops, scope, lim, t_{nb}, t_{exp}, nonce, del, sig \rangle. \tag{4}$$

记 $\mathcal{C}(T)$ 为已绑定任务主体、通过规范编码与基础解析的候选能力集。每次动作仲裁都必须在当前环境中重新检查其中至少一项能力，而不能因为早先任务步骤已通过而沿用结论。

动作在环境 e 下满足能力，当且仅当：

$$\mathrm{CapOK}(a,T,e) \Leftrightarrow \exists c \in \mathcal{C}(T): S(c) \wedge B(a,T,c) \wedge O(a,c) \wedge L(a,c,e) \wedge F(c,e). \tag{5}$$

其中 cid, ver, rev 是稳定能力标识、版本与撤销版本；$S(c)$ 验证规范签名，$B(a,T,c)$ 绑定动作、任务主体与对象，O, L, F 分别验证操作/范围、额度/委派和时窗/nonce/撤销。令 $e = \langle e_R, e_C, e_P, e_V, t\rangle$ 冻结风险、能力、策略、验证状态与判定时间。定义 $D(T,a,e) = \neg\mathrm{CapOK}(a,T,e) \vee ExplicitForbid \vee BindingMismatch \vee HardFail$；$U(T,a,e)$ 为必要属性、风险快照或验证材料缺失、过期或冲突；$H(T,a,e) = ReviewReq(T,a,e) \wedge \neg AuthOK(T,a,e)$。策略集合 P 输出：

$$\mathrm{Decide}(T,a,e) = \begin{cases} \text{Deny}, & D(T,a,e); \\ \text{Escalate}, & \neg D(T,a,e) \wedge [U(T,a,e) \vee H(T,a,e)]; \\ \text{Permit}, & \neg D(T,a,e) \wedge \neg U(T,a,e) \wedge \neg H(T,a,e). \end{cases} \tag{6}$$

三个分支两两互斥且完备：禁止优先，未知进入升级，只有材料完整且所需批准有效时才可放行。$ReviewReq$ 在策略含人工复核规则、$R_{inh}(T,a,e_R) > \tau_{hard}$ 或 $R_{res}(T,a,e_R) > \tau_{auto}$ 时为真，且 $0 \le \tau_{auto} \le \tau_{hard} \le 1$。AuthOK 只统计签名恢复后的唯一批准主体；每份批准签署同一 baseDecisionHash，绑定批准 nonce 与过期时间。baseDecisionHash 承诺网络域、任务版本、计划、动作、能力、策略、风险及验证摘要；规范排序后的批准形成 approvalSetHash，最终 decisionHash 再承诺决定、原因和有效期。批准 nonce 只在副作用保留事务中一次性消费，避免签名摘要循环和只读检查提前消耗授权。

3.3 风险、验证等级与路由

设风险向量 $r = (r_{asset}, r_{priv}, r_{irr}, r_{ext}, r_{unc}, r_{chain}) \in [0,1]^6$，分别刻画资产暴露、隐私、不可逆性、工具副作用、模型不确定性和跨链/预言机依赖。风险环境 e_R 冻结估值源、数据等级、目标链与区块、合约代码、模型/工具版本、riskSnapshotHash、riskModelVersion、observedAt、validUntil。先不计入未验证的控制效果，得到固有风险：

$$R_{inh}(T,a,e_R) = \sigma\left(\beta_0^{m_R} + \sum_i w_i^{m_R} r_i + \sum_{i<j} \gamma_{ij}^{m_R} r_i r_j\right), \quad \sigma(x) = \frac{1}{1+e^{-x}}. \tag{7}$$

强制保证下限由 R_{inh} 决定，不得因自报控制而降低。只有配置摘要匹配、签名有效且未过期的控制量才进入 $R_{res}(T,a,e_R) = \sigma\big(\mathrm{logit} R_{inh}(T,a,e_R) - \lambda^{m_R} m_{verified}(T,a,e_R)\big)$，其中 $\lambda^{m_R} \ge 0$、$m_{verified}(T,a,e_R) \ge 0$；控制未验证时取零。$\mathcal{A}(T)$ 是计划中可能产生外部效果的动作集，若

$\mathcal{A}(T) = \varnothing$，两类任务风险均定义为 0。对路径 j，令 $R^j_{inh} = \max_{a\in\mathcal{A}_j} R_{inh}(T,a,e_j)$、$R^j_{res} = \max_{a\in\mathcal{A}_j} R_{res}(T,a,e_j)$；若 $\mathcal{A}_j = \varnothing$，两者也定义为 0。前者固定验证下限，后者仅进入成本、时延、监控与人工升级权衡；二者不是客观事故概率。

将 L0–L4 保留为展示套餐：L0 为模式与日志，L1 增加确定规则、签名和模拟，L2 增加独立数据源或冗余执行，L3 增加经证明隔离环境或多方见证，L4 对可形式化关系使用密码证明。实际门禁不把它们视为单一总序，而为每类声明选择验证组合：

$$\begin{aligned}\mathcal{V}_F(T,j,e_j) = \{\mathbf{v}\in\mathcal{V}(T,j) \mid & CfgValid(\mathbf{v},e_j) = 1,\\ & \forall k\in K_{T,j}: Cvg(\mathbf{v},k)\ge\tau_k(R^j_{inh}),\\ & Ind(\mathbf{v},k)\ge q_k(R^j_{inh}),\ Req(\mathbf{v},k) = 1\},\\ & \mathbf{v}^*_j(T,e_j)\in \arg\min_{\mathbf{v}\in\mathcal{V}_F(T,j,e_j)} Cost_j(\mathbf{v}).\end{aligned} \tag{8}$$

$K_{T,j}$ 是路径关键声明集合，Cvg 与 Ind 分别是 Coverage 与 Independence 的紧凑记号；Req 要求密码证明、同区块状态或账户不变量等硬条件全部显式存在且为真。τ_k, q_k 对固有风险单调不减。CfgValid 要求注册表签名、配置摘要、风险/声明绑定和时效有效；任一关键声明缺失都不能被其他声明高分抵消，没有可行配置时必须升级或拒绝。

候选路径 $j\in\mathcal{J}(T)$ 组合模型、节点、工具、链和验证器。以成本 C_j、尾时延 L_j 与证据支持的可信度代理 Q_j 建立多目标路由：

$$j^* = \arg\min_{j\in\mathcal{J}(T)} \left[\alpha(R^j_{res})\widehat{C}_j + \beta(R^j_{res})\widehat{L}_j - \gamma(R^j_{res})\widehat{Q}_j\right], \tag{9}$$

$$\begin{aligned}\text{s.t.}\ & C_j\le b_c,\ G_j\le b_g,\ M_j\le b_m,\ L_j\le D - t_0,\\ & Q_j(\mathbf{v}^*_j)\ge Q_{min}(R^j_{inh}),\quad \mathbf{v}^*_j\in\mathcal{V}_F(T,j,e_j),\\ & \forall a\in\mathcal{A}_j:\ Decide(T,a,e_j) = \text{Permit}.\end{aligned} \tag{10}$$

风险升高时，Q_{min} 与各关键声明的覆盖、独立性阈值不得下降。路径选定后，系统将 $verificationConfigHash = \text{CfgHash}(v^*_{j^*})$ 冻结进任务版本；状态迁移和回执只接受同一配置的 verificationResultHash。若可行集为空，正确行为是拒绝、延迟或拆分，而不是放松硬约束。共享同一 RPC、模型或索引的验证器不得计为完全独立。

3.4 状态、证据与回执

抽象状态机记为 $M = (S,\Sigma,\delta,s_0,F)$。迁移只在保护条件成立时发生：

$$(s_{t+1},q_{t+1}) = \delta\big((s_t,q_t),event_t\big) \Leftrightarrow G_t = 1 \wedge CAS(id,q_t,q_t+1) = 1. \tag{11}$$

其中 q_t 为状态版本，G_t 同时检查合法边、taskVersionHash、计划/动作、未过期决定、能力版本/撤销、实际路径验证配置与最终性；串行化事务中的 CAS 防止并发事件都从同一旧状态成功。抽象层要求副作用前存在同一任务版本的有效授权，完成态前存在按实际路径配置 $\mathbf{v}_{j^*}^*$ 执行且结果通过的声明级验证记录。第 5 章给出的具体运行态是这一抽象的工程细化；两者不是两套互相竞争的状态机。

对证据集合 $E_T = \{e_1, \ldots, e_n\}$ 采用任务域绑定的规范化 Merkle 承诺。记 d_N 为 HZB 网络域，H_P, H_{in} 为计划摘要与输入根，d_C 为目标链或外系统域；域上下文 χ_i 依次编码 networkDomain、tenant、H(T_v)、planHash、inputRoot、subjectRef、chainDomain、window：

$$h_i = H\big(\chi_i \,\|\, \mathrm{canon}(e_i)\big), \qquad R_T^E = \mathrm{MerkleRoot}(h_1, \ldots, h_n). \tag{12}$$

原文可留在分级授权存储，链上仅按需锚定固定长度根。规范编码、算法标识和叶序必须固定，否则同一证据会产生不同根。可复用证据保留全局内容摘要，但每次 EvidenceUse 必须重新绑定租户、任务版本、计划、输入、声明对象、链域与时间窗，防止把其他任务中的真实证据拼接到当前任务。承诺可检测事后替换，却不证明原始内容真实。

为避免各类回执扩展后失去关联，采用共同头部加 kind 判别载荷的类型化信封。令 d_L, d_S 分别为 HZB/REC/LEAF/v1 与 HZB/REC/SIG/v1 的域标签：

$$\begin{gathered}
Rec = \langle h, p_k, \rho_R, \Sigma_k \rangle, \quad h = \langle k, v, d_N, s, \mathcal{P}, t, \nu, \psi_k \rangle, \\
\big((q_i, x_i)\big)_{i=1}^{m} = Sort_{UTF8}\big(Leaves(h, p_k)\big), \\
\ell_i = H\big(d_L \,\|\, v \,\|\, k \,\|\, utf8(q_i) \,\|\, canon(x_i)\big), \quad \rho_R = MerkleRoot(\ell_1, \ldots, \ell_m), \\
\Sigma_k = \{Sign_{alg,u,r,kid}\big(H(d_S \,\|\, v \,\|\, d_N \,\|\, k \,\|\, u \,\|\, r \,\|\, kid \,\|\, alg \,\|\, t_s \,\|\, \rho_R)\big): \\
(u, r, kid, alg, t_s) \in ReqSigners(k, p_k, \psi_k)\}.
\end{gathered} \tag{13}$$

h 对应 kind、schemaVersion、networkDomain、subjectHash、parentRoots、observedAt、nonce、signerPolicyHash；p_k 必须属于该版本的唯一载荷变体。execution 载荷含状态、成本、任务/意图/计划、输入/输出、能力/策略、证据/状态及可选外部效果；contribution 绑定 sourceApp、建设任务版本、验收槽、Builder、工件、权利主张、父闭包根与 claimNullifier；validation 绑定 contribution 根、验证策略、轮次、报告集合、裁决和申诉截止；dispute、settlement 与 anchor 分别绑定争议状态、义务终态和锚定交易。

Leaves 不含 receiptRoot/signatures，路径采用转义 JSON Pointer 并按 UTF-8 字节序排序；字符串统一 NFC，整数使用固定单位，集合先去重排序而有序数组保留原序。parentRoots 是唯一的直接父集合，拒绝未知字段、缺失字段及用 null 冒充不适用字段；空集合使用版本化空

根。ψ_k 固定类型签名策略，Σ_k 按主体、角色与历史密钥版本签署同一 ρ_R。执行器、Builder、计入法定人数的 Validator、争议裁决者、独立结算控制器和链域 Anchor Attestor 的类型签名不可由通用服务签名替代。

其完整性谓词为：

$$Intact(Rec) \Leftrightarrow VariantOK_k \wedge CanonRootOK_k \wedge DigestOK_k \wedge ParentPolicyOK_k \wedge StateOK_k \wedge FreshOK_k \wedge RoleSigOK_k. \quad (14)$$

VariantOK 检查 kind 与载荷变体；CanonRootOK 重算规范叶及 receiptRoot；DigestOK 重算该类型的内部摘要；ParentPolicyOK 检查父集合、闭包及任务/建设域绑定；StateOK 检查状态与最终性；FreshOK 检查时窗和 nonce；RoleSigOK 使用签名时刻的历史密钥注册表，核对主体、角色、类型、策略、密钥版本与撤销时间。因而 Validator 签名不能冒充结算授权，统一“回执引擎公钥”也不能成为单点信任根。

业务层另行报告证据覆盖率：

$$Cov(T) = \frac{\sum_{k \in K_T} \omega_k \, \mathbf{1}[\exists e \in E_T : \mathrm{Verify}(e,k) = 1]}{\sum_{k \in K_T} \omega_k}. \quad (15)$$

于是“回执完整”与“关键声明已覆盖”成为两个独立判断。有效签名不能把低覆盖率结果显示为“完全验证”；同样，覆盖率高也不意味着业务决定必然正确。

3.5 HZB 双增长引擎与 Proof of Build 形式化

HZB 的第一增长引擎是可被真实业务调用的 Web3 AI 基础设施：Web3 LLM 负责专业理解与规划，Developer API/SDK 负责开放能力，Smart Contract Tools 负责形成可测试工件，AI Agent 负责执行受控工作流，HIVM 则统一任务、权限、验证和回执语义。第二增长引擎是全球共建网络：Proof of Build 与 Venture Cell 把项目需求拆成可验收任务，让 Builders、Agent、Validators 与生态伙伴围绕里程碑协作。前者产生任务级 HIVM Execution Receipt，后者只在任务级证据闭合后形成建设级证明；因此“调用成功”不自动等于“贡献有效”。

一项 PoB 建设任务及其贡献、建设证明定义为：

$$\begin{gathered} BT = \langle d_N, bid, v_B, src, iss, g_B, M, A, E^{req}, \mathbf{P}_B, R^{rights}, Bud, D, \nu_B, \Sigma_B \rangle, \\ C = Rec\left[k = contribution,\ p_k = \langle src, bid, v_B, mid, slot, builder, R_A, R_E, R_{cl}, R_{rights}, N_C \rangle\right], \\ BP = \langle bpid, H(BT), mid, \rho_C, \rho_V, R_{cl}, K_c, N_C, aState, t_{appeal}, \rho_D \rangle. \end{gathered} \quad (16)$$

其中 d_N 是 HZB 网络域，v_B 冻结建设任务及 Venture Cell 模板版本，src 是已签名 sourceApp，$\mathbf{P}_B = (VPol, SPol)$ 固定 verificationPolicy、settlementPolicy。C 的 $slot$ 是冻结的

claimSlotId，R_A, R_E, R_{cl} 分别为工件根、证据根和全部可达父 execution 根的规范闭包根，N_C 是贡献空值符；ρ_C, ρ_V 分别是唯一 contribution 与 validation 回执根，mid 在 BP 中固定里程碑。BP 不是 Receipt，其稳定根单独定义为 bpRoot = H("HZB/PoB/BP/v1", networkDomain, canon(BP))，规范对象排除 bpRoot 自身。稳定验收键 K_c = $H(\text{"HZB/PoB/claim/v1"}, d_N, bid, v_B, mid, slot)$，空值符 N_C = $H\big(\text{"HZB/PoB/nullifier/v1"}, d_N, R_A, R_{rights}\big)$。前者保证一个验收槽只有一个最终 BP，后者阻止同一工件在同一权利主张下换身份或里程碑重复申领；合法多人贡献需在单一 contribution 中冻结参与者与分配关系。

建设证明有效性不能由单一哈希或单一模型评分决定。令 V 为唯一 validation 回执、Γ 为唯一性注册表；b_C, b_{DAG}, b_V 分别检查 contribution 类型签名、父 DAG 完整/无环/同域/成功/最终性，以及 validation 目标、策略、协议轮次和法定人数。b_{appeal} 要求申诉关闭且生效裁决为 ACCEPT，其余量表示绑定、证据、冲突、撤销、争议和唯一性条件：

$$\begin{aligned}\mathrm{PoBValid}(BT, C, V, BP, \Gamma) = {} & b_{bind} \wedge b_C \wedge b_{DAG} \wedge b_V \wedge b_{evidence} \\ & \wedge b_{quorum} \wedge b_{conflict} \wedge b_{appeal} \wedge b_{nr} \wedge b_{nd} \\ & \wedge b_{accepted} \wedge b_{unique}(\Gamma, K_c, N_C, BP).\end{aligned} \tag{17}$$

其中 $b_{accepted}$ 明确要求 aState(BP)=FINAL_ACCEPTED，b_{unique} 要求 claimKey 与 nullifier 在 Γ 中均绑定当前 BP 根且无竞争记录。任一项为假，状态只能停留在补证、争议、拒绝或保留，不能进入结算。模型和自动预检可以生成风险提示、测试摘要或覆盖率，但不能成为高价值任务的唯一裁决者。令 O 为尚未清偿的冻结义务，结算采用失效安全门：

$$\begin{aligned}K_s &= H\big(\text{"HZB/SETTLEMENT/v1"}, d_N(BT), bid(BT), v_B(BT), mid(O), oid(O), v_O(O)\big), \\ a^* &= \begin{cases} Release\Big(FrozenRelease\big(O, \alpha(BP)\big)\Big), & \mathrm{PoBValid}(BT, C, V, BP, \Gamma) = 1; \\ Refund\big(FrozenRefund(O)\big), & \mathrm{RejectBound}(BT, V, BP, O) = 1; \\ \bot, & \text{otherwise}, \end{cases} \\ \mathrm{Settle}(BT, C, V, BP, \Gamma, O) &= \begin{cases} \mathrm{ApplyOnce}(K_s, O, a^*), & a^* \neq \bot \wedge Outstanding(O); \\ \mathrm{Hold}(O), & \text{otherwise}. \end{cases}\end{aligned} \tag{18}$$

式中 RejectBound 核验最终拒绝裁决及 BP、建设任务、里程碑与冻结义务绑定。K_s 仅依赖冻结义务身份及版本，Release 与 Refund 竞争同一主键；branch、收款方、资产、金额、chainNonce、feePolicyHash、callHash 作为冻结意图保存，首次写入后不得切换分支。授权 nonce、能力额度、链 nonce 与义务必须在同一串行化事务中保留；不确定是否提交时进入 UNKNOWN，只允许重播完全相同的原始交易。最终清偿、settlement receipt 与 final outbox 全有或全无，使“已释放 + 已退款 + 未清偿 = 原冻结义务”保持不变。分配函数 α 只能使用发布时冻结的规则；本模型不预设 HZB 已发行代币。

4 五层总体架构

4.1 分层原则与数据流

HZB 总体架构遵循“链下形成智能，链上建立信任与价值闭环”；HIVM 在这一边界内采用“类型化对象跨层、自由文本止于认知边界”的运行原则。入口请求生成 intentHash，任务图生成 planHash，证据集合生成 evidenceRoot，策略决定生成 decisionHash，外部交易生成 txHash 和最终区块引用。五类摘要共同进入回执，形成从意图到效果的关联链；PoB 再把符合里程碑的一组回执根绑定到 artifactRoot 与验证裁决。任何摘要变化都使后续授权或建设证明失效，不能被解释成无关紧要的文本修订。

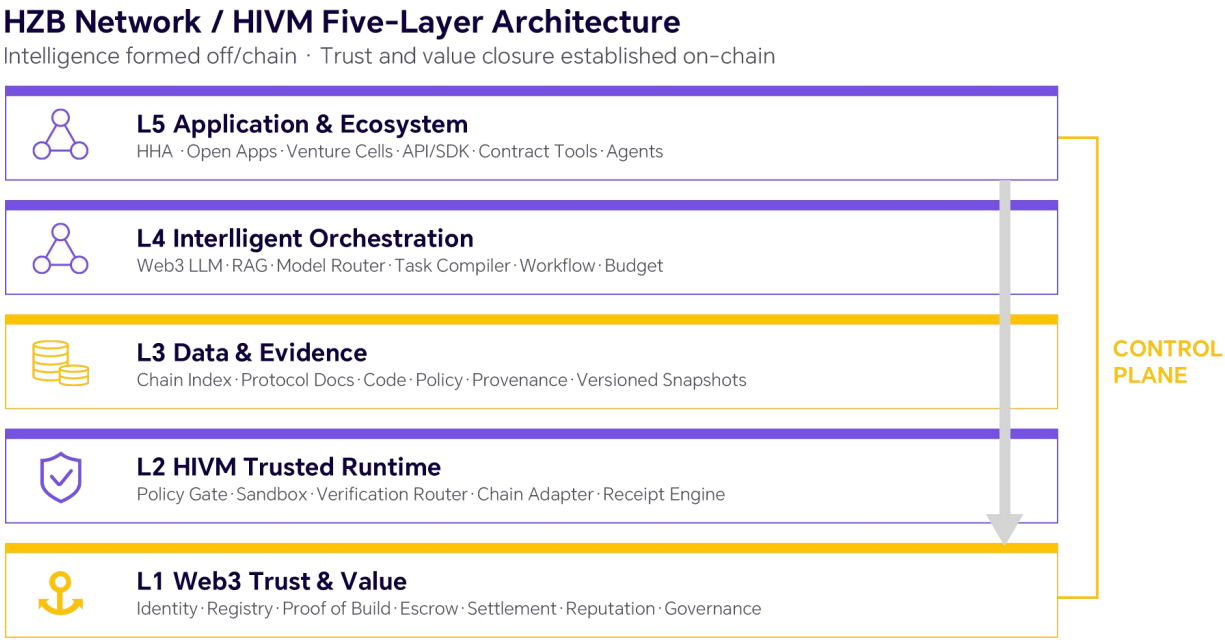

图 1 HIVM 五层总体架构

表 2 五层职责、输入输出与禁止事项

层次	主要职责	关键输出	明确禁止
L5 应用与生态入口	AI Hub、API/SDK、合约工具、Agent、HHA 与外部生态	规范化意图、主体、Task Package	入口直接广播交易或绕过 PoB 规则
L4 智能编排	Web3 LLM、RAG、任务编译、模型/工具路由与预算	冻结计划、结构化候选工件	模型自行扩大能力或写审批字段
L3 数据与证据	数据来源、版本、时效、验证与证据图	证据引用、冲突报告、Merkle 根	把检索命中等同于真实
L2 HIVM 可信运行	策略门、状态机、沙箱、验证、补偿与回执	决策摘要、事件流、任务回执	绕过授权或掩盖部分失败

层次	主要职责	关键输出	明确禁止
L1 Web3 信任与价值	身份、Registry、Credential、Escrow、Settlement、Reputation 与 Governance	承诺、交易、PoB 状态、结算与信誉事件	由 HIVM 或 PoB 替代底层共识与专业判断

4.2 应用入口与智能编排

应用层对外划分 /chat、/chain-data、/contracts、/agents、/evidence、/build-tasks 与 /proofs 等接口域。它们是能力边界而非 URL 分类：/chain-data 默认只读，/contracts 生成的写动作必须进入授权，/agents/{id}/run 不得绕过 Policy Engine，/build-tasks 不能事后降低验收或知识产权条件，/proofs 只能聚合已绑定任务和里程碑的证据。身份可组合钱包地址、DID 与可验证凭证，但身份声明、资格证明、实时授权、签名有效性与建设信誉仍是不同层次[21-23]。

编排层包含 Task Compiler、Model Router 和 Tool Runtime。Task Compiler 消除关键歧义、展开依赖并插入模拟/验证/授权节点；Model Router 按能力、成本、隐私和可复现性选择模型；Tool Runtime 在短时能力和沙箱内调用工具。三者均不能持有通用签名权。MCP 与 A2A 可作为能力暴露和 Agent 互操作适配器[34-35]，但协议互通不会自动建立授权或信任。

4.3 数据证据与可信运行

数据层在内容进入模型前绑定来源、许可、观察时间、区块高度、内容摘要、索引和验证器版本；模型产物反向引用其输入证据。Verification Router 按声明类型分派验证器：链状态要求独立 RPC 在同一 blockHash 上一致；合约主张组合字节码摘要、ABI、静态规则与仿真；外部事实检查签名和时效；难形式化语义进入独立复核或人工评审。反对证据必须保留，不能只存多数结果。

可信运行层由 Policy Engine、受控执行循环和 Receipt Engine 构成。策略输出 PERMIT/DENY/ESCALATE、原因码、限额、审批人数、有效期和策略版本；只有 PERMIT 可进入执行，ESCALATE 必须转入等待授权。运行时只追加事件并以比较—交换推进状态；Receipt Engine 规范化意图、计划、证据、策略、签名与链上结果。该层不执行底层共识，只保证提交给价值层的动作满足预定前置条件。

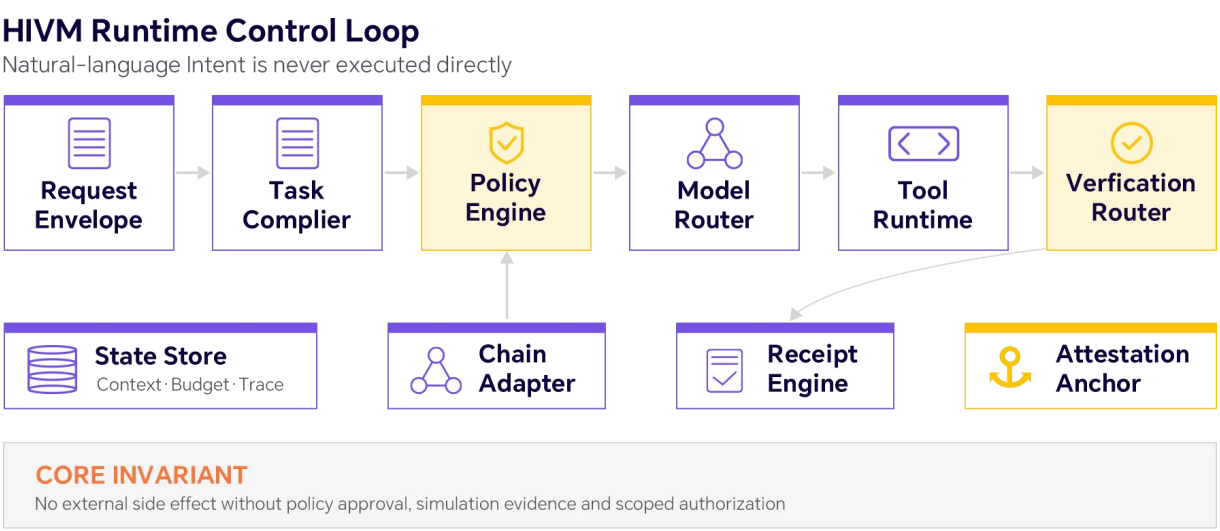

图 2 HIVM 运行时控制闭环

4.4 Web3 价值层与七组件边界

价值层包含既有公链、Layer2、联盟链、钱包、账户抽象、预言机和合约。Chain Adapter 统一 RPC、费用、nonce、交易编码、确认深度和最终性差异，但不决定“是否应该交易”。七个核心组件是职责而非强制微服务：Task Compiler 不签名，Model Router 不裁决资产风险，Tool Runtime 不扩权，Policy Engine 不伪造证据，Verification Router 不代替用户授权，Chain Adapter 不改变冻结调用，Receipt Engine 不把失败包装成成功。这样即使部署形态变化，安全边界仍可测试。

5 运行时、指令、双路径、状态机与回执

5.1 八类指令与中间表示

HIVM 定义八类上层指令：READ 读取事实；ANALYZE 分类和推理；GENERATE 产生结构化候选；SIMULATE 在固定上下文预测效果；VALIDATE 检查模式、证据和不变量；AUTHORIZE 绑定策略与主体批准；EXECUTE 提交外部状态改变；ATTEST 生成回执。它们是任务中间表示，不是底层操作码，也不要求模型步骤逐字确定。

表 3　指令、副作用与最低门禁

指令	默认副作用	主要工件	最低控制
READ	无	带来源事实	能力、时效、租户隔离
ANALYZE	无	风险/解释	模型版本、输入引用
GENERATE	无	严格模式候选	Schema、预算、禁止自由文本直达执行
SIMULATE	无真实提交	状态差分、调用轨迹	固定 chainId、区块与代码摘要
VALIDATE	无	报告与冲突	声明类型匹配、独立性与阈值
AUTHORIZE	产生权限状态	绑定 callHash 的批准	策略、主体、限额、时窗
EXECUTE	有	外部事务标识	冻结参数、幂等、签名域、最终性规则
ATTEST	本地签名无外部副作用	步骤/任务回执	规范编码、签名、最小披露；链上锚定另走受权 EXECUTE(anchor)

每条指令还带输入引用、输出模式、依赖、预算、超时、重试策略、风险和幂等键。GENERATE 的自由文本不得成为 EXECUTE 参数；写动作必须先结构化，再经过模拟和验证。AUTHORIZE 固定 chainId、to、value、calldataHash、deadline 等调用摘要，任何字段变化都要求新计划和新授权。

5.2 认知路径与价值路径

认知路径为 READ → ANALYZE → GENERATE → SIMULATE → VALIDATE，允许并行检索、多模型比较和候选淘汰，但没有外部状态修改权。价值路径为 AUTHORIZE → EXECUTE → ATTEST，其中只有明确 Permit 且绑定冻结参数的动作可进入 EXECUTE；ATTEST 首先是本地规范编码与签名，若要链上锚定，必须再生成独立的 EXECUTE(anchor) 动作并重新过策略

门。路径依“能否产生外部副作用”划分：链上只读 eth_call 属认知路径，向托管服务发出提现则属价值路径。

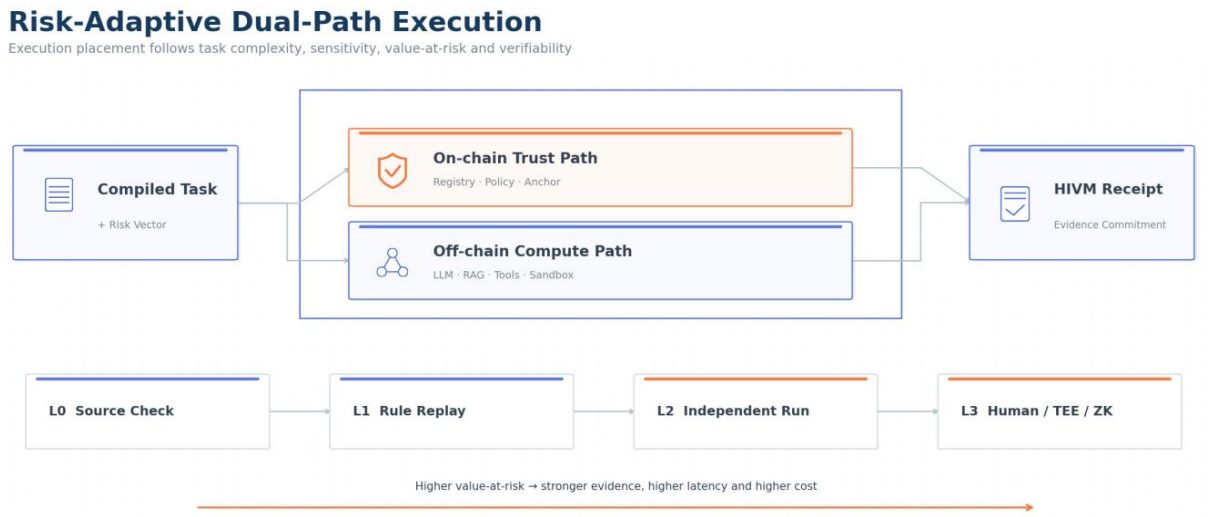

图 3 HIVM 认知路径与价值路径

两条路径在验证—授权门汇合。风险较低的只读任务可以在认知路径生成离线回执；资产、治理或跨系统动作必须进入价值路径。策略拒绝不是系统故障，而是可审计结果；验证冲突不能由模型用自然语言“解释掉”，应补充证据、升级人工或失败关闭。

5.3 事件驱动状态机

工程状态将推理与副作用分开：CREATED → COMPILED → COGNITIVE_RUNNING，无副作用任务可直接形成 ATTESTED。外部效果若 Policy Gate 首次返回 ALLOW，可直接物化为 AUTHORIZED；只有 ESCALATE 进入 WAITING_AUTH，在批准后再到 AUTHORIZED。随后统一经过 EFFECT_EXECUTING → SUBMITTED → CONFIRMED → ATTESTED，未授权认知运行绝不能进入效果态。不可恢复错误进入 FAILED；已有部分效果则进入 COMPENSATING。

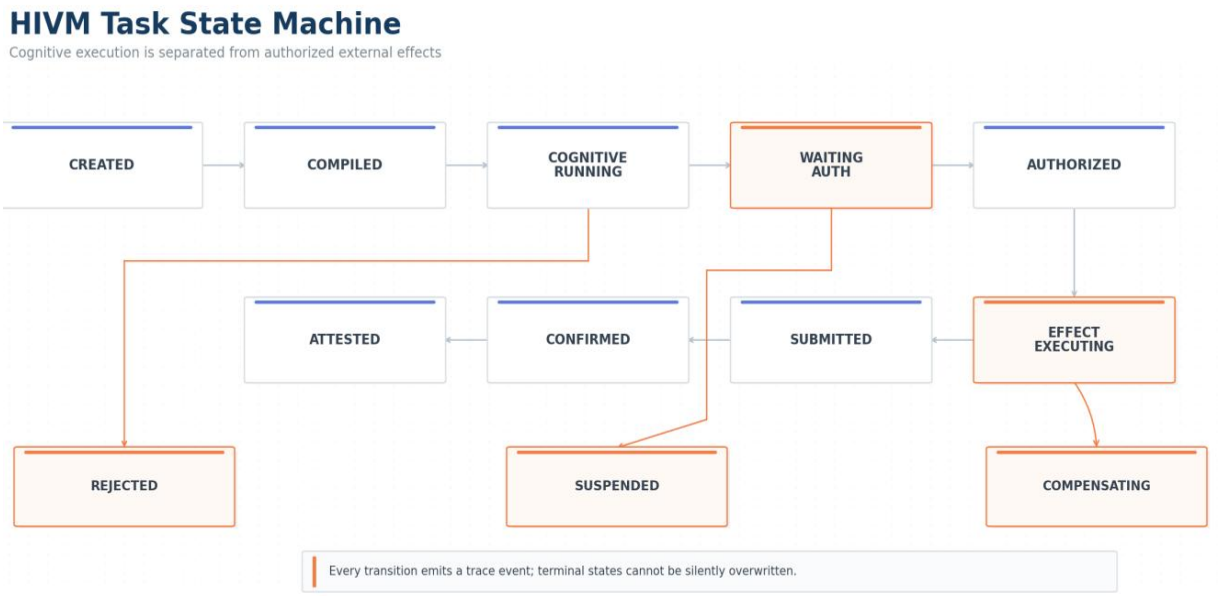

图 4 HIVM 任务状态机

系统采用“至少一次调度、指令级效果至多一次”的组合语义，合法多步任务可包含多个不同效果。队列可重复投递，执行器以任务版本、指令、动作和链域生成幂等键，并与累计能力额度同事务保留。接管工作器先查询外部状态，不得假设前一工作器未广播。并行 READ 按指令 ID 规范排序，避免完成顺序改变 evidenceRoot。

5.4 失败、补偿与最终性

运行时区分四类失败：瞬时网络中断和限流可指数退避重试；模式错误、策略拒绝和签名无效是确定性失败；广播后未及时获得回执属于不确定提交，必须先按 sender、nonce 和交易池查询；跨域流程已有一步成功则是部分效果失败，需要补偿或人工处置。补偿不是回滚区块链，通常是经过完整 SIMULATE → VALIDATE → AUTHORIZE → EXECUTE 的新反向动作。没有安全补偿的流程应降低额度、使用托管或拆分阶段。

CONFIRMED 在达到应用定义的最终性前可以是软状态。Chain Adapter 观察规范链，区块引用失效时撤销软确认、重新验证并保留重组事件。经济终局任务只有达到预设确认深度或网络最终性信号后，才可产生 ATTESTED/SUCCESS。因此“RPC 返回交易哈希”“交易进入区块”和“达到业务最终性”是三个不同事件。

5.5 分级验证与三层回执

表 4　验证层次及不能推出的结论

层次	可检查对象	代表机制	不能推出
完整性	字节是否与承诺一致	哈希、Merkle 路径	内容真实
来源	谁签发或采集	签名、VC、数据来源记录	签发者没有错误
环境	哪个度量在何环境运行	RATS/TEE/EAT[17-18,20]	程序语义正确、无侧信道
规则执行	结构化关系是否满足	规则复算、zk/zkML[19]	模型目标合理
结果复核	多源或重执行是否一致	独立 RPC、冗余模型、人工	一致即真
业务确认	主体是否理解并接受确定动作	人可读摘要、多签、审批	决策必然获利或合法

回执分三层：步骤回执记录单条指令的输入输出摘要、版本和耗时；任务回执聚合状态迁移、策略、授权、验证和执行结果；锚定回执记录任务回执根在特定链上的交易与区块。ATTEST 失败不会使已确认交易消失，调用方应看到“执行已确认、证明待完成”的双状态。链上事件只证明指定摘要被锚定，回执中的事实仍需由被披露证据与验证方法支持。

5.6 从 HIVM Execution Receipt 到 Proof of Build

HIVM Execution Receipt 与 PoB Build Record 不可互换。前者记录一次受控执行的输入、版本、策略、授权、事件与证据完整性；后者记录建设任务、里程碑、贡献主体、验证决定、争议状态与结算结果。二者通过 buildTaskId、hivmTaskId、receiptHash、evidenceRoot 关联，但有效执行回执不能自动推出交付合格、权属成立或应当结算。

PoB 将验收与结算状态分离。BP 经 DRAFT → VALIDATING → ACCEPTED_PENDING_APPEAL/REJECTED_PENDING_APPEAL ↔ DISPUTED → FINAL_ACCEPTED/FINAL_REJECTED；独立 Settlement Record 再由 UNSETTLED 进入 FINAL_RELEASED 或 FINAL_REFUNDED。结算不得把 BP 的 FINAL_ACCEPTED 改写为 SETTLED。一个 Build Task 可编译成多个 HIVM 任务，而验收、申诉、信誉与结算属于 PoB 应用协议，不是 HIVM 内核新指令。PoB 不是 Proof of Work，也不是 HZB 底层共识机制；它证明的仅是“指定交付在冻结策略与已声明假设下通过验收”。

6 代码逻辑与 API 参考实现

6.1 接口契约与端到端调用

本章代码均为**参考实现/接口示意**，用于让形式模型落到类型、控制流和状态约束；持久化事务、HSM/KMS、并发隔离、密钥轮换、跨链异常和完整错误处理。

表 5　建议 API 域及其安全语义

API 域	典型操作	返回对象	强制边界
/chat	创建意图、补充澄清	taskId、编译状态	不接受私钥，不直接执行写操作
/chain-data	查询余额、事件、区块	带 blockHash 的 EvidenceRef	默认只读，多源与时效策略
/contracts	解析 ABI、构造与模拟调用	候选调用、状态差分	模拟结果不可视为授权
/agents	注册能力、运行/暂停任务	能力句柄、状态事件	任务级最小权限、可撤销
/evidence	查询回执、验证包含证明	分层披露的证据包	租户隔离、最小披露与访问日志

入口 POST /chat/tasks 只创建 CREATED 任务；客户端用 GET /agents/tasks/{taskId} 观察状态或订阅事件。编译后的候选调用必须通过 /contracts/simulate 与 /evidence/verify，高风险任务在 /agents/tasks/{taskId}/approvals 接收绑定 callHash 的批准，随后才可请求执行。所有写请求携带客户端幂等键；服务端返回稳定任务标识，而不是在超时后暗中创建第二个任务。traceId 可按 OpenTelemetry 与 W3C Trace Context 跨服务传播[24-25]，但追踪标识不是授权凭据，也不得承载敏感内容。

6.2 任务、能力、证据与回执数据结构

代码清单 1　任务与回执核心数据结构（TypeScript 参考实现）

```typescript
// 接口：生产实现、规范编码库和字段级披露策略。
type Hash = `0x${string}`;
type Op = "READ" | "ANALYZE" | "GENERATE" | "SIMULATE" |
          "VALIDATE" | "AUTHORIZE" | "EXECUTE" | "ATTEST";
type ReceiptKind = "execution" | "contribution" | "validation" |
                   "dispute" | "settlement" | "anchor";
type TaskState = "CREATED" | "COMPILED" | "COGNITIVE_RUNNING" |
                 "WAITING_AUTH" | "AUTHORIZED" | "EFFECT_EXECUTING" |
                 "SUBMITTED" | "CONFIRMED" |
                 "COMPENSATING" | "FAILED" | "ATTESTED";

interface Budget { moneyMicros: bigint; gas: bigint; modelCalls: number; }
interface Capability {
  capabilityId: Hash; version: number; revocationVersion: number;
  issuer: string; subject: string; object: string; operations: Op[];
  scope: Record<string, unknown>; limits: Record<string, bigint>;
  notBefore: number; expiresAt: number; nonce: string; signature: string;
```

```
}
interface Instruction {
  id: string; op: Op; dependsOn: string[]; inputRefs: Hash[];
  capabilityRef: Hash; capabilityVersion: number;
  capabilityRevocationVersion: number; capabilityHash: Hash; actionHash: Hash;
  outputSchema: string; risk: "read" | "low" | "high";
  timeoutMs: number; maxAttempts: number; idempotencyKey: Hash;
}
interface Task {
  taskId: string; principal: string; version: number; versionHash: Hash;
  intentHash: Hash; planHash: Hash; verificationConfigHash: Hash;
  finalityPolicyHash: Hash;
  verificationResultHash?: Hash; stateVersion: number;
  activeActionHash?: Hash; activeDecisionHash?: Hash;
  authorizationHash?: Hash;
  instructions: readonly Instruction[]; capabilities: readonly Capability[];
  budget: Budget; deadline: number; state: TaskState;
}
interface EvidenceRef {
  digest: Hash; source: string; observedAt: number; verifier: string;
  chainId?: number; blockHash?: Hash; disclosure: "public" | "audit" | "restricted";
}
interface ReceiptHeader<K extends ReceiptKind> {
  kind: K; schemaVersion: string; networkDomain: string; subjectHash: Hash;
  parentRoots: readonly Hash[]; observedAt: number; nonce: string;
  signerPolicyHash: Hash;
}
interface ExecutionPayload {
  taskHash: Hash; intentHash: Hash; planHash: Hash;
  inputRoot: Hash; outputHash: Hash; capabilityHash: Hash;
  decisionHash: Hash; evidenceRoot: Hash; stateRoot: Hash;
  buildContextHash?: Hash; status: "SUCCESS" | "FAILED" | "PARTIAL";
  cost: Budget;
  chainRef?: { chainId: number; txHash: Hash;
               finalityRef: string; actionHash: Hash };
}
interface ContributionPayload {
  sourceApp: string; buildTaskId: string; buildTaskVersion: number;
  milestoneId: string; claimSlotId: string; builderId: string;
  artifactRoot: Hash; evidenceRoot: Hash; parentClosureRoot: Hash;
  rightsClaimHash: Hash; claimNullifier: Hash;
}
interface ValidationPayload {
  pobContextHash: Hash; contributionReceiptRoot: Hash;
  verificationPolicyHash: Hash; validationRoundId: string;
  claimSetRoot: Hash; reportSetRoot: Hash; verifierSetHash: Hash;
  verdict: "ACCEPT" | "REJECT"; appealDeadline: number;
}
interface DisputePayload {
  pobContextHash: Hash; targetReceiptRoot: Hash;
  reasonRoot: Hash; resolutionRoot?: Hash;
  authorizesRevalidation: boolean;
  state: "OPEN" | "UPHELD" | "OVERTURNED";
}
interface SettlementPayload {
  buildProofRoot: Hash; validationReceiptRoot: Hash;
  obligationId: string; obligationVersion: number; settlementKey: Hash;
  branch: "RELEASE" | "REFUND"; recipient: string;
  asset: string; amount: bigint; callHash: Hash;
  executionReceiptRoot: Hash; finalityRef: string;
}
interface AnchorPayload {
  anchoredReceiptRoot: Hash; chainId: number; txHash: Hash; finalityRef: string;
}
```

```ts
type ReceiptPayloadMap = {
  execution: ExecutionPayload; contribution: ContributionPayload;
  validation: ValidationPayload; dispute: DisputePayload;
  settlement: SettlementPayload; anchor: AnchorPayload;
};
interface ReceiptSignature {
  subject: string; role: string; keyId: string;
  algorithm: string; signedAt: number; value: string;
}
type UnsignedReceipt<K extends ReceiptKind> = {
  header: ReceiptHeader<K>; payload: ReceiptPayloadMap[K];
};
type ReceiptEnvelope = {
  [K in ReceiptKind]: UnsignedReceipt<K> & {
    receiptRoot: Hash; signatures: readonly ReceiptSignature[];
  };
}[ReceiptKind];

function computeReceiptRoot<K extends ReceiptKind>(r: UnsignedReceipt<K>): Hash {
  strictVariantCheck(r.header.kind, r.header.schemaVersion, r.payload);
  requireCanonicalParentSet(r.header.parentRoots);
  const leaves = canonicalTypedLeaves(r)
    .sort((a, b) => compareUtf8(a.path, b.path));
  return merkleRoot(leaves.map(({ path, value }) => sha256(canonicalEncode([
    "HZB/REC/LEAF/v1", r.header.schemaVersion, r.header.kind,
    utf8(path), value
  ]))));
}

function receiptSigningMessage(r: ReceiptEnvelope,
                               signer: ReceiptSignature): Hash {
  return sha256(canonicalEncode([
    "HZB/REC/SIG/v1", r.header.schemaVersion, r.header.networkDomain,
    r.header.kind, signer.subject, signer.role, signer.keyId,
    signer.algorithm, signer.signedAt, r.receiptRoot
  ]));
}
```

关键字段由不同权威来源写入：身份层写主体，授权服务写能力，策略仓库写决定，Chain Adapter 写交易/最终性，PoB 协议写建设载荷；模型不得把同名文本伪装成系统元数据。解码器先依据 kind/schemaVersion 选择唯一载荷模式，再重算规范 Merkle 根；验证者依据 signerPolicyHash 与历史注册表核对类型化签名，不能用当前密钥状态改写历史责任。

6.3 Task Compiler

Task Compiler 类似安全工作流编译器。它先区分用户明示字段与推断字段，再按能力目录展开任务 DAG；若链、资产、数量、接收方、滑点或数据用途等关键字段缺失，则生成澄清计划。对可能产生副作用的计划，编译器自动插入模拟、验证、授权与证明节点，并做支配关系和预算静态检查。

代码清单 2　Task Compiler（参考伪代码）

```
function COMPILE(intent, principal, policySnapshot, capabilityCatalog):
    n = normalize(intent)
    require schemaValid(n) and authenticated(principal)
```

```
    if missingCriticalFields(n):
        return clarificationPlan(n.missing, markInferredFields(n))

    g = new DAG()
    facts    = g.add(READ,     n.dataRequirements)
    analysis = g.add(ANALYZE,  refs(facts))
    proposal = g.add(GENERATE, refs(analysis), strictSchema=n.outputSchema)

    if n.mayChangeExternalState:
        sim  = g.add(SIMULATE, refs(proposal), fixedContext=n.chainContext)
        val  = g.add(VALIDATE, refs(facts, proposal, sim), claims=n.claims)
        auth = g.add(AUTHORIZE, canonicalHash(proposal),
                     refs(val, policySnapshot))
        exec = g.add(EXECUTE, refs(proposal, auth))
        g.add(ATTEST, refs(exec, val, auth))
    else:
        val = g.add(VALIDATE, refs(facts, proposal), claims=n.claims)
        g.add(ATTEST, refs(proposal, val))

    require acyclic(g)
    require everyExecuteDominatedByAuthorize(g)
    require everyHighRiskInputHasFreshEvidence(g)
    require worstCaseCost(g) <= n.budget
    bindCapabilitiesAndIdempotency(g, principal, n.intentHash, capabilityCatalog)
    return freeze(g), hash(canonicalEncode(g))
```

编译结果一经授权即冻结。价格、nonce、区块或代码摘要变化若影响调用，应生成新 planHash 并回到 VALIDATE/AUTHORIZE；不能在旧签名下静默替换参数。静态检查不能证明业务逻辑无误，但可以证明某些危险图形在构造阶段不可出现，例如没有授权支配节点的 EXECUTE。

6.4 状态机、幂等与效果控制

代码清单 3　合法状态迁移与幂等执行（Python 参考实现）

```python
# 接口示意：store 需提供事务、租约、版本比较和只追加事件日志。
ALLOWED = {
    "CREATED": {"COMPILED", "FAILED"},
    "COMPILED": {"COGNITIVE_RUNNING", "FAILED"},
    "COGNITIVE_RUNNING": {"WAITING_AUTH", "AUTHORIZED", "ATTESTED", "FAILED"},
    "WAITING_AUTH": {"AUTHORIZED", "FAILED"},
    "AUTHORIZED": {"EFFECT_EXECUTING", "FAILED"},
    "EFFECT_EXECUTING": {"SUBMITTED", "COMPENSATING", "FAILED"},
    "SUBMITTED": {"CONFIRMED", "COMPENSATING"},
    "CONFIRMED": {"SUBMITTED", "ATTESTED", "COMPENSATING"},
    "COMPENSATING": {"FAILED", "ATTESTED"},
}

def guard_transition(snapshot, target, event, now):
    pair = (snapshot.state, target)
    if event.get("taskVersionHash") != snapshot.version_hash:
        return False
    if pair == ("CREATED", "COMPILED"):
        return event.get("planHash") == snapshot.plan_hash and event.get("compilerSigOk")
is True
    if pair == ("COMPILED", "COGNITIVE_RUNNING"):
        return event.get("leaseOk") is True and event.get("budgetReserved") is True
    if pair == ("COGNITIVE_RUNNING", "WAITING_AUTH"):
        return event.get("decision", {}).get("effect") == "ESCALATE"
```

```
    if pair in {("COGNITIVE_RUNNING", "AUTHORIZED"),
                ("WAITING_AUTH", "AUTHORIZED")}:
        return (event["decision"]["effect"] == "ALLOW"
                and event["decision"]["validUntil"] > now
                and verify_policy_decision(event["decision"],
                    snapshot.version_hash, event["actionHash"])
                and event["decision"].get("approvalSetHash") is not None)
    if pair == ("AUTHORIZED", "EFFECT_EXECUTING"):
        auth = event["authorization"]
        return (verify_typed_authorization(auth, snapshot.principal, now)
                and auth["actionHash"] == event["actionHash"]
                and auth["planHash"] == snapshot.plan_hash
                and auth["decisionHash"] == event["decision"]["decisionHash"]
                and event["decision"]["effect"] == "ALLOW"
                and event["decision"]["validUntil"] > now
                and auth["capabilityHash"] == event["capabilityHash"]
                and auth["capabilityVersion"] == event["capabilityVersion"]
                and auth["revocationVersion"] == event["revocationVersion"]
                and event["activeActionHash"] == event["actionHash"]
                and event["activeDecisionHash"] ==
                    event["decision"]["decisionHash"]
                and event["authorizationHash"] == hash_canonical(auth)
                and auth["expiresAt"] > now)
    if pair == ("EFFECT_EXECUTING", "SUBMITTED"):
        return bool(event.get("effectKey") and event.get("submissionRef"))
    if pair == ("SUBMITTED", "CONFIRMED"):
        return (event.get("confirmationOk") is True
                and event.get("finalityPolicyHash") == snapshot.finality_policy_hash)
    if pair == ("CONFIRMED", "SUBMITTED"):  # 软确认被重组撤销
        return event.get("reorgProofOk") is True and event.get("wasFinal") is False
    if pair == ("CONFIRMED", "ATTESTED"):
        return (event["finality"]["ok"] is True
                and verification_guard(snapshot, event["verification"], now))
    if pair in {("COGNITIVE_RUNNING", "ATTESTED"),
                ("COMPENSATING", "ATTESTED")}:
        return (verification_guard(snapshot, event["verification"], now)
                and (snapshot.state != "COMPENSATING"
                     or event.get("compensationFinalityOk") is True))
    if target == "COMPENSATING":
        return (event.get("failureEvidence") is not None
                and event.get("compensationAuthorizationOk") is True)
    if target == "FAILED":
        return (event.get("failureEvidence") is not None
                and (snapshot.state != "EFFECT_EXECUTING"
                     or event.get("provenNotSubmitted") is True))
    return False

def verification_guard(snapshot, verification, now):
    return (verify_verification_result(verification)
            and verification["ok"] is True
            and verification["configHash"] == snapshot.verification_config_hash
            and verification["validUntil"] > now
            and (snapshot.verification_result_hash is None or
                 snapshot.verification_result_hash ==
                    verification["verificationResultHash"]))

def transition(store, task_id, expected_state, expected_task_version_hash,
               expected_state_version, target, event, now):
    with store.serializable(task_id) as tx:
        snapshot = tx.load_current()
        if (snapshot.state != expected_state or
                snapshot.version_hash != expected_task_version_hash or
                snapshot.state_version != expected_state_version):
```

```
            raise RuntimeError("state/version conflict")
        if target not in ALLOWED.get(snapshot.state, set()):
            raise ValueError(f"illegal transition: {snapshot.state} -> {target}")
        if not guard_transition(snapshot, target, event, now):
            raise PermissionError("transition guard rejected")
        first_verification_hash = (
            event["verification"]["verificationResultHash"]
            if target == "ATTESTED" and
               snapshot.verification_result_hash is None else None)
        return tx.append_transition(target, event,
            expected_state_version=expected_state_version,
            set_verification_result_hash=first_verification_hash)

NONTERMINAL_EFFECT_STATES = {"RESERVED", "DISPATCHING", "SUBMITTED", "UNKNOWN"}

def canonical_action_projection(network_domain, action):
    return {"domain": "HZB/ACTION/v1", "networkDomain": network_domain,
        "chainId": action.chainId, "chainDomain": action.chainDomain,
        "chainNonce": action.chainNonce, "feePolicyHash": action.feePolicyHash,
        "sender": action.sender.lower(), "to": action.to.lower(),
        "calldataHash": hash_canonical(action.calldata),
        "valueWei": action.valueWei,
        "assetDeltasRoot": hash_canonical(action.assetDeltas),
        "deadline": action.deadline, "op": action.op,
        "objectRef": action.objectRef,
        "scopeHash": hash_canonical(action.scope),
        "usageDeltaRoot": hash_canonical(action.usageDelta),
        "authorizationNonce": action.nonce}

def run_instruction(ctx, task_id, ins, action, decision,
                    authorization, now, reserve_hook=None):
    task = ctx.store.read_task(task_id)
    ctx.schemas.validate_action(ins.op, action)
    action_hash = hash_canonical(
        canonical_action_projection(ctx.network_domain, action))
    if action_hash != ins.actionHash:
        raise PermissionError("authorized and dispatched action differ")
    effect_key = hash_parts(ctx.network_domain, task.version_hash,
                            ins.id, action_hash, action.chainDomain)
    if ins.idempotencyKey != effect_key:
        raise PermissionError("idempotency key mismatch")
    with ctx.store.atomic_effect(effect_key) as tx:
        existing = tx.get_effect()
        if existing and existing.state == "COMMITTED":
            return existing.result
        if existing and existing.state in NONTERMINAL_EFFECT_STATES:
            return tx.defer_reconciliation(existing)  # 不再次计额或 dispatch
        if existing:
            raise RuntimeError("terminal effect requires a new task version")
        if decision.validUntil <= now:
            raise PermissionError("policy decision expired")
        authorization_hash = hash_canonical(authorization)
        tx.assert_task_state(task_id, task.version_hash, "EFFECT_EXECUTING")
        tx.assert_effect_binding(task_id, task.version_hash,
            action_hash, decision.decisionHash, authorization_hash)
        tx.assert_permit_decision(
            decision, task.version_hash, action_hash, now=now)
        tx.assert_typed_authorization(authorization, task.principal,
            action_hash, decision.decisionHash, now=now)
        tx.consume_approval_nonces_once(
            task.version_hash, decision.approvalBindings,
            expected_hash=decision.approvalSetHash, now=now)
        reservation = tx.reserve_capability(
```

```
            capability_id=ins.capabilityRef,
            capability_version=ins.capabilityVersion,
            capability_hash=ins.capabilityHash,
            subject=task.principal, operation=action.op,
            object_ref=action.objectRef, scope=action.scope,
            usage_delta=action.usageDelta, nonce=action.nonce,
            revocation_version=ins.capabilityRevocationVersion,
        )  # 同一事务内重验签名、范围、撤销、nonce、额度并累计消费
        chain_nonce_reservation = tx.reserve_chain_nonce_once(
            chain_id=action.chainId, sender=action.sender,
            chain_nonce=action.chainNonce, action_hash=action_hash)
        if reserve_hook is not None:
            reserve_hook(tx, reservation)  # 业务义务/空值符与能力配额同事务保留
        tx.mark_dispatching(
            reservation, chain_nonce_reservation, action_hash)
    try:
        result = ctx.dispatch(action, deadline_ms=ins.timeoutMs)
        ctx.schemas.validate(ins.outputSchema, result)
        return ctx.store.record_dispatch_result(reservation, result)
    except ctx.DefinitePreDispatchError:
        ctx.store.release_if_not_submitted(
            reservation, chain_nonce_reservation)
        raise
    except Exception:
        ctx.store.mark(reservation, "UNKNOWN")
        return ctx.reconcile_frozen_effect(effect_key)  # 未查明前不得重发
```

stateVersion 与不可变 taskVersionHash 分开递增；verificationConfigHash、finalityPolicyHash 均进入 versionHash，状态快照映射为 verification_config_hash、finality_policy_hash。首次完成事件可在守卫验证后于同一 CAS 写入 verificationResultHash，后续迁移只接受同一结果。Policy Gate 与执行器共享版本化 canonicalActionProjection，避免批准摘要和实际调用选择不同字段。进入效果态时持久化 activeActionHash、activeDecisionHash、authorizationHash，效果保留事务逐项重验。effectKey = H(networkDomain ‖ taskVersionHash ‖ instructionId ‖ actionHash ‖ chainDomain) 同时就是编译后的 idempotencyKey。atomic_effect 原子保留 effectKey、累计能力额度和链 nonce，并重验任务态、决定、批准 nonce、能力及授权 nonce；已有 RESERVED/DISPATCHING/SUBMITTED/UNKNOWN 只进入对账，不重复计额或发送。

6.5 Policy Gate

策略引擎输入是由可信组件组装的主体、动作、证据与批准，而不是完整提示词。输出除三值决定外，还包含原因码、策略版本和决定摘要；接口枚举 ALLOW 对应形式模型的 Permit，DENY/ESCALATE 同理。EIP-712 的类型化签名有助于域隔离和人可读授权，但规范本身不包含重放保护，仍须绑定 nonce、deadline、chainId 和 verifying contract[26]；合约账户可通过 ERC-1271 适配签名验证[27]，跨链交易域还应考虑 EIP-155[28]。

代码清单 4　TypeScript Policy Gate（参考实现）

```ts
type Tri = "PASS" | "FAIL" | "MISSING";
type Approval = {
  signer: string; baseDecisionHash: Hash; nonce: Hash;
  expiresAt: number; signature: string;
};
type ApprovalBinding = {
  signer: string; baseDecisionHash: Hash; nonce: Hash;
  expiresAt: number; signature: string; digest: Hash;
};
type GateInput = {
  networkDomain: string; taskVersionHash: Hash; planHash: Hash;
  action: { actionHash: Hash; chainId: number; chainDomain: string;
            chainNonce: bigint; feePolicyHash: Hash;
            sender: string; to: string; calldataHash: Hash;
            valueWei: bigint; assetDeltasRoot: Hash; deadline: number;
            op: Op; objectRef: string; scopeHash: Hash;
            usageDeltaRoot: Hash; authorizationNonce: string };
  frozen: { capabilityBindingHash: Hash; policyConfigHash: Hash;
            riskSnapshotHash: Hash; verificationConfigHash: Hash };
  capability: { bindingHash: Hash; version: number; valid: boolean };
  policy: { version: string; configHash: Hash };
  risk: { snapshotHash: Hash; riskResultHash: Hash; modelVersion: string;
          inherent: number; residual: number;
          observedAt: number; validUntil: number };
  verification: { configHash: Hash; riskSnapshotHash: Hash;
    verificationResultHash: Hash;
    reportSetHash: Hash; claims: Tri; simulation: Tri;
    observedAt: number; validUntil: number };
  approvals: readonly Approval[];
};
type GateDecision = {
  effect: "ALLOW" | "DENY" | "ESCALATE"; reasons: readonly string[];
  policyVersion: string; policyConfigHash: Hash;
  baseDecisionHash: Hash; approvalSetHash: Hash;
  approvalBindings: readonly ApprovalBinding[];
  decisionHash: Hash; validUntil: number;
};

function canonicalActionProjection(networkDomain: string,
                                   a: GateInput["action"]) {
  return { domain: "HZB/ACTION/v1", networkDomain,
    chainId: a.chainId, chainDomain: a.chainDomain,
    chainNonce: a.chainNonce, feePolicyHash: a.feePolicyHash,
    sender: a.sender.toLowerCase(), to: a.to.toLowerCase(),
    calldataHash: a.calldataHash, valueWei: a.valueWei,
    assetDeltasRoot: a.assetDeltasRoot, deadline: a.deadline,
    op: a.op, objectRef: a.objectRef, scopeHash: a.scopeHash,
    usageDeltaRoot: a.usageDeltaRoot,
    authorizationNonce: a.authorizationNonce };
}

function compareUtf8Bytes(a: string, b: string): number {
  const A = new TextEncoder().encode(a.normalize("NFC"));
  const B = new TextEncoder().encode(b.normalize("NFC"));
  for (let i = 0; i < Math.min(A.length, B.length); i++)
    if (A[i] !== B[i]) return A[i] - B[i];
  return A.length - B.length;
}

function policyGate(x: GateInput, now: number): GateDecision {
  const deny: string[] = [], incomplete: string[] = [];
  const computedActionHash = sha256(canonicalJson(
```

```
  canonicalActionProjection(x.networkDomain, x.action))) as Hash;
const computedRiskResultHash = sha256(canonicalJson({
  domain: "HZB_HIVM_RISK_RESULT_V1", snapshotHash: x.risk.snapshotHash,
  modelVersion: x.risk.modelVersion, inherent: x.risk.inherent,
  residual: x.risk.residual, observedAt: x.risk.observedAt,
  validUntil: x.risk.validUntil
})) as Hash;
const computedVerificationResultHash = sha256(canonicalJson({
  domain: "HZB_HIVM_VERIFY_RESULT_V1",
  configHash: x.verification.configHash,
  riskSnapshotHash: x.verification.riskSnapshotHash,
  reportSetHash: x.verification.reportSetHash,
  claims: x.verification.claims, simulation: x.verification.simulation,
  observedAt: x.verification.observedAt,
  validUntil: x.verification.validUntil
})) as Hash;
if (computedActionHash !== x.action.actionHash)
  deny.push("ACTION_HASH_MISMATCH");
if (computedRiskResultHash !== x.risk.riskResultHash)
  deny.push("RISK_RESULT_HASH_MISMATCH");
if (computedVerificationResultHash !==
    x.verification.verificationResultHash)
  deny.push("VERIFICATION_RESULT_HASH_MISMATCH");
if (!ALLOWLISTED_CHAINS.has(x.action.chainId)) deny.push("CHAIN_DENIED");
if (!ALLOWLISTED_CONTRACTS.has(x.action.to.toLowerCase())) deny.push("TARGET_DENIED");
if (x.action.deadline <= now) deny.push("ACTION_EXPIRED");
if (x.action.valueWei > DAILY_LIMIT_WEI) deny.push("LIMIT_EXCEEDED");
if (!x.capability.valid || x.capability.bindingHash !==
    x.frozen.capabilityBindingHash) deny.push("CAPABILITY_BINDING_FAILED");
if (x.policy.configHash !== x.frozen.policyConfigHash)
  deny.push("POLICY_BINDING_FAILED");
if (x.risk.snapshotHash !== x.frozen.riskSnapshotHash)
  deny.push("RISK_BINDING_FAILED");
if (x.verification.configHash !== x.frozen.verificationConfigHash)
  deny.push("VERIFICATION_BINDING_FAILED");
if (x.verification.riskSnapshotHash !== x.risk.snapshotHash)
  deny.push("VERIFICATION_RISK_BINDING_FAILED");
if (x.verification.claims === "FAIL" ||
    x.verification.simulation === "FAIL") deny.push("HARD_CONDITION_FAILED");
if (x.verification.claims === "MISSING" ||
    x.verification.simulation === "MISSING")
  incomplete.push("VERIFICATION_MISSING");
if (x.risk.validUntil <= now) incomplete.push("RISK_SNAPSHOT_STALE");
if (x.verification.validUntil <= now) incomplete.push("VERIFICATION_STALE");
if (x.risk.inherent < 0 || x.risk.inherent > 1 ||
    x.risk.residual < 0 || x.risk.residual > 1)
  deny.push("RISK_RESULT_INVALID");

const reviewRequired = policyRequiresReview(x) ||
  x.risk.inherent > HARD_REVIEW_THRESHOLD ||
  x.risk.residual > AUTO_EXECUTION_THRESHOLD;
const baseReasons = [...new Set([
  ...deny, ...incomplete, ...(reviewRequired ? ["REVIEW_REQUIRED"] : [])
])].sort(compareUtf8Bytes);
const baseEffect = deny.length ? "DENY" :
  (incomplete.length || reviewRequired) ? "ESCALATE" : "ALLOW";
const baseValidUntil = Math.min(
  x.action.deadline, x.risk.validUntil, x.verification.validUntil);
const baseDecisionHash = sha256(canonicalJson({
  domain: "HZB_HIVM_POLICY_BASE_V1", networkDomain: x.networkDomain,
  taskVersionHash: x.taskVersionHash, planHash: x.planHash,
  actionHash: computedActionHash,
  capabilityBindingHash: x.capability.bindingHash,
  capabilityVersion: x.capability.version,
```

```
    policyVersion: x.policy.version, policyConfigHash: x.policy.configHash,
    riskSnapshotHash: x.risk.snapshotHash,
    riskResultHash: computedRiskResultHash,
    verificationConfigHash: x.verification.configHash,
    verificationResultHash: computedVerificationResultHash,
    reportSetHash: x.verification.reportSetHash,
    baseEffect, baseReasons, validUntil: baseValidUntil
  })) as Hash;

  const bySigner = new Map<string, Approval & { digest: Hash }>();
  if (!deny.length && !incomplete.length && reviewRequired) {
    for (const a of x.approvals) {
      if (a.baseDecisionHash !== baseDecisionHash || a.expiresAt <= now) continue;
      const recovered = normalizePrincipal(recoverTypedApprovalSigner({
        networkDomain: x.networkDomain, baseDecisionHash,
        nonce: a.nonce, expiresAt: a.expiresAt
      }, a.signature));
      if (recovered !== normalizePrincipal(a.signer) ||
          !APPROVER_SET.has(recovered) ||
          !approvalNonceUnused(x.taskVersionHash, recovered, a.nonce)) continue;
      const digest = sha256(canonicalJson({
        signer: recovered, baseDecisionHash,
        nonce: a.nonce, expiresAt: a.expiresAt,
        signatureHash: sha256(a.signature)
      })) as Hash;
      const old = bySigner.get(recovered);
      if (!old || a.expiresAt > old.expiresAt ||
          (a.expiresAt === old.expiresAt && digest < old.digest))
        bySigner.set(recovered, { ...a, signer: recovered, digest });
    }
  }
  const approvals = [...bySigner.values()]
    .sort((a, b) => compareUtf8Bytes(a.signer, b.signer));
  const approvalBindings: ApprovalBinding[] = approvals.map(a => ({
    signer: a.signer, baseDecisionHash, nonce: a.nonce,
    expiresAt: a.expiresAt, signature: a.signature, digest: a.digest
  }));
  const approvalSetHash = sha256(canonicalJson(approvalBindings)) as Hash;
  const authOK = !reviewRequired ||
    quorumSatisfied(approvals.map(a => a.signer), x.action);
  const effect: GateDecision["effect"] = deny.length ? "DENY" :
    (incomplete.length || (reviewRequired && !authOK)) ? "ESCALATE" : "ALLOW";
  const reasons = [...new Set(effect === "DENY" ? deny :
    effect === "ESCALATE" ? [...incomplete,
      ...(reviewRequired && !authOK ? ["APPROVAL_QUORUM_MISSING"] : [])] : [])]
    .sort(compareUtf8Bytes);
  const validUntil = reviewRequired && authOK
    ? Math.min(baseValidUntil, ...approvals.map(a => a.expiresAt))
    : baseValidUntil;
  const decisionHash = sha256(canonicalJson({
    domain: "HZB_HIVM_POLICY_DECISION_V1", baseDecisionHash,
    approvalSetHash, effect, reasons, validUntil
  })) as Hash;
  return { effect, reasons, policyVersion: x.policy.version,
    policyConfigHash: x.policy.configHash, baseDecisionHash,
    approvalSetHash, approvalBindings, decisionHash, validUntil };
}
```

生产策略应由可审查的确定性规则或策略语言承载，并对策略变更进行签名、影子回放和分权审批。批准者签署不含批准集合的 baseDecisionHash，Policy Gate 再把含签名证明的规范 approvalBindings 纳入最终决定；副作用控制器重算 approvalSetHash、复核签名与时效后原

子消费其中 nonce。Gate 中的 verificationResultHash 是 Verification Aggregator 对各 Router 结果、模拟结果、配置、报告集合及有效期形成的汇总摘要，必须按代码中的同一字段集重算，不能由调用方另行填写有效期。拒绝事件同样进入回执，以区分模型失败、证据不足、策略禁止和人工未批准。

6.6 Verification Router

Verification Router 先按声明类型选择验证器，再评估报告新鲜度、权重、共同依赖和硬条件。加权一致不是事实真值；高风险声明可要求同一 blockHash、至少一个密码证明或账户不变量全部成立。若没有适配验证器，结果必须是 NO_VERIFIER，而不是默认通过。

代码清单 5　Python Verification Router（参考实现）

```python
from dataclasses import dataclass, asdict

@dataclass(frozen=True)
class Claim:
    kind: str
    digest: str
    context_hash: str
    not_before: int

@dataclass(frozen=True)
class RegistryEntry:
    verifier_id: str
    public_key: str
    allowed_kinds: tuple[str, ...]
    fault_domain: str
    weight: float
    valid_from: int
    valid_until: int
    revoked_at: int | None = None

@dataclass(frozen=True)
class RegistrySnapshot:
    version: str
    root: str
    valid_until: int
    entries: tuple[RegistryEntry, ...]
    governance_signature: str

@dataclass(frozen=True)
class VerificationConfig:
    config_hash: str
    registry_root: str
    claim_kind: str
    context_hash: str
    risk_snapshot_hash: str
    allowed_verifiers: tuple[str, ...]
    score_threshold: float
    min_reports: int
    min_weight: float
    required_independence: int
    required_hard_conditions: tuple[str, ...]
    max_age_seconds: int
```

```
    max_clock_skew_seconds: int
    valid_until: int
    policy_key_id: str
    policy_signature: str

class VerificationRouter:
    def __init__(self, registry, config, clients,
                 registry_trust_key, config_trust_key):
        entries = [asdict(e) for e in sorted(
            registry.entries, key=lambda x: x.verifier_id)]
        if len({e.verifier_id for e in registry.entries}) != len(registry.entries):
            raise ValueError("REGISTRY_ID_DUPLICATE")
        if any(e.weight <= 0 for e in registry.entries):
            raise ValueError("REGISTRY_WEIGHT_INVALID")
        if hash_canonical(entries) != registry.root:
            raise ValueError("REGISTRY_ROOT_MISMATCH")
        signed_registry = hash_canonical({
            "version": registry.version, "root": registry.root,
            "validUntil": registry.valid_until
        })
        if not verify_signature(registry_trust_key, signed_registry,
                                registry.governance_signature):
            raise ValueError("REGISTRY_SIGNATURE_INVALID")
        config_body = asdict(config)
        config_signature = config_body.pop("policy_signature")
        config_body.pop("config_hash")
        if hash_canonical(config_body) != config.config_hash:
            raise ValueError("CONFIG_HASH_MISMATCH")
        if not verify_signature(config_trust_key, config.config_hash,
                                config_signature):
            raise ValueError("CONFIG_SIGNATURE_INVALID")
        if config.registry_root != registry.root:
            raise ValueError("CONFIG_REGISTRY_MISMATCH")
        safe_bounds = (0 <= config.score_threshold <= 1 and
            config.min_reports >= 1 and config.min_weight > 0 and
            config.required_independence >= 1 and
            config.max_age_seconds > 0 and
            config.max_clock_skew_seconds >= 0)
        if not safe_bounds:
            raise ValueError("CONFIG_SAFETY_BOUNDS_INVALID")
        self.registry, self.config, self.clients = registry, config, clients
        self.entries = {e.verifier_id: e for e in registry.entries}

    def _result(self, ok, reason, reports, now, **metrics):
        report_hashes = sorted(r[2] for r in reports)
        report_set_hash = domain_merkle_root(
            "HZB/VERIFY/REPORTS/v1", report_hashes)
        expiry_candidates = [self.config.valid_until,
                             self.registry.valid_until]
        for reg, report, _ in reports:
            expiry_candidates.extend([
                reg.valid_until, report["expiresAt"],
                report["observedAt"] + self.config.max_age_seconds])
            if reg.revoked_at is not None:
                expiry_candidates.append(reg.revoked_at)
        valid_until = min(expiry_candidates)
        body = {"ok": ok, "reason": reason, "configHash":
                self.config.config_hash,
                "riskSnapshotHash": self.config.risk_snapshot_hash,
                "reportSetHash": report_set_hash,
                "observedAt": now, "validUntil": valid_until, **metrics}
        return {**body, "reports": [r[1] for r in reports],
                "verificationResultHash": hash_canonical(body)}
```

```
    def verify(self, claim: Claim, risk_snapshot_hash: str, now: int) -> dict:
        cfg = self.config
        if risk_snapshot_hash != cfg.risk_snapshot_hash:
            return self._result(False, "RISK_SNAPSHOT_BINDING_MISMATCH", [], now)
        if claim.kind != cfg.claim_kind or claim.context_hash != cfg.context_hash:
            return self._result(False, "CLAIM_BINDING_MISMATCH", [], now)
        if now >= cfg.valid_until or now >= self.registry.valid_until:
            return self._result(False, "CONFIG_OR_REGISTRY_STALE", [], now)
        selected = []
        for verifier_id in sorted(set(cfg.allowed_verifiers)):
            reg = self.entries.get(verifier_id)
            active = reg and reg.valid_from <= now < reg.valid_until and (
                reg.revoked_at is None or now < reg.revoked_at)
            if active and claim.kind in reg.allowed_kinds and verifier_id in self.clients:
                selected.append(reg)
        if not selected:
            return self._result(False, "NO_VERIFIER", [], now)

        accepted = []
        for reg in selected:
            try:
                report = self.clients[reg.verifier_id].verify(
                    claim, cfg.config_hash)
                body = {name: report[name] for name in (
                    "verifierId", "claimDigest", "configHash", "contextHash",
                    "verdict", "observedAt", "expiresAt", "hardConditions", "nonce")}
                bound = (body["verifierId"] == reg.verifier_id and
                    body["claimDigest"] == claim.digest and
                    body["configHash"] == cfg.config_hash and
                    body["contextHash"] == claim.context_hash and
                    body["verdict"] in ("PASS", "FAIL") and
                    verify_signature(reg.public_key, hash_canonical(body),
                                     report["signature"]))
                fresh = (claim.not_before <= body["observedAt"] <=
                    now + cfg.max_clock_skew_seconds and
                    now - body["observedAt"] <= cfg.max_age_seconds and
                    now < body["expiresAt"])
                if not bound or not fresh:
                    continue
                digest = hash_canonical({"body": body,
                                         "signature": report["signature"]})
                accepted.append((reg, report, digest))
            except (KeyError, TypeError, ValueError):
                continue
        if not accepted:
            return self._result(False, "NO_VALID_REPORT", [], now)

        passed = [x for x in accepted if x[1]["verdict"] == "PASS"]
        total_weight = sum(x[0].weight for x in accepted)
        pass_weight = sum(x[0].weight for x in passed)
        score = pass_weight / total_weight if total_weight else 0.0
        domains = {x[0].fault_domain for x in passed}
        hard_ok = all(
            any(x[1]["hardConditions"].get(name) is True for x in passed)
            and not any(x[1]["hardConditions"].get(name) is False
                        for x in accepted)
            for name in cfg.required_hard_conditions
        )
        ok = (len(accepted) >= cfg.min_reports and
              pass_weight >= cfg.min_weight and
              score >= cfg.score_threshold and
              len(domains) >= cfg.required_independence and hard_ok)
        reason = "PASS" if ok else "THRESHOLD_OR_HARD_CONDITION_FAILED"
        return self._result(ok, reason, accepted, now, score=score,
```

```
            passWeight=pass_weight, faultDomains=sorted(domains),
            hardConditionsOk=hard_ok)
```

注册表而非报告提供 weight、faultDomain、公钥，且注册表与配置本身必须验签和重算摘要；报告绑定 claim、context 与 config，并通过过去/未来双边新鲜度检查。缺配置、缺验证器、报告不足、权重不足、故障域不足或硬条件冲突均失败关闭。规范 reportSetHash 与 verificationResultHash 进入 Policy Gate、状态守卫和回执，防止路由后静默替换配置或报告集合。

6.7 Receipt Anchor

链上锚定只保存固定长度摘要与时间，不公开完整提示、证据或隐私数据。下例加入 EIP-712 域、任务绑定、重复检查和低 s 检查，仍只是说明协议语义；实际部署需要经审计库、多证明者治理、密钥轮换、暂停和批量 Merkle 根机制[29-31]。

代码清单 6　Solidity 执行回执锚定（参考实现）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

/// @notice 接口示意；未审计，不可直接用于真实资产或生产治理。
contract HIVMReceiptAnchor {
    bytes32 private constant DOMAIN_TYPEHASH = keccak256(
        "EIP712Domain(string name,string version,uint256 chainId,address
verifyingContract)"
    );
    bytes32 private constant RECEIPT_TYPEHASH = keccak256(
        "Receipt(bytes32 taskIdHash,bytes32 receiptHash,bytes32 evidenceRoot)"
    );
    uint256 private constant HALF_ORDER =
      0x7fffffffffffffffffffffffffffffff5d576e7357a4501ddfe92f46681b20a0;
    address public immutable attestor;
    bytes32 public immutable DOMAIN_SEPARATOR;
    mapping(bytes32 => uint256) public anchoredAt;

    event ReceiptAnchored(bytes32 indexed taskIdHash,
        bytes32 indexed receiptHash, bytes32 evidenceRoot, uint256 timestamp);

    constructor(address trustedAttestor) {
        require(trustedAttestor != address(0), "zero attestor");
        attestor = trustedAttestor;
        DOMAIN_SEPARATOR = keccak256(abi.encode(DOMAIN_TYPEHASH,
            keccak256("HIVMReceiptAnchor"), keccak256("1"),
            block.chainid, address(this)));
    }

    function anchor(bytes32 taskIdHash, bytes32 receiptHash,
        bytes32 evidenceRoot, bytes calldata sig) external {
        require(anchoredAt[receiptHash] == 0, "already anchored");
        bytes32 structHash = keccak256(abi.encode(
            RECEIPT_TYPEHASH, taskIdHash, receiptHash, evidenceRoot));
        bytes32 digest = keccak256(abi.encodePacked(
            "\x19\x01", DOMAIN_SEPARATOR, structHash));
        require(recover(digest, sig) == attestor, "invalid attestation");
        anchoredAt[receiptHash] = block.timestamp;
```

```
        emit ReceiptAnchored(taskIdHash, receiptHash, evidenceRoot, block.timestamp);
    }

    function recover(bytes32 h, bytes calldata sig) internal pure returns (address) {
        require(sig.length == 65, "bad signature length");
        bytes32 r; bytes32 s; uint8 v;
        assembly {
            r := calldataload(sig.offset)
            s := calldataload(add(sig.offset, 32))
            v := byte(0, calldataload(add(sig.offset, 64)))
        }
        if (v < 27) v += 27;
        require((v == 27 || v == 28) && uint256(s) <= HALF_ORDER, "bad sig");
        return ecrecover(h, v, r, s);
    }
}
```

锚定事件能证明某摘要在特定链、合约和时间由指定证明者确认；它不能证明摘要内的外部事实真实，也不能证明建设验收、付款权利或 Venture Cell 法律权利成立。审计者仍需获得证据叶、Merkle 路径、策略版本和状态事件，才能重算端到端关联。

6.8 Proof of Build 聚合与结算参考实现

PoB 作为 HIVM 之上的应用协议，只消费类型化回执，不绕过内核门禁。HHA 和其他入口均提供已签名的 sourceApp，不因“创世应用”身份获得额外验证、财库或策略权限。下面的 Settlement Effect Controller 是 run_instruction/atomic_effect 在冻结义务场景下的专用实现，复用同一动作摘要、Policy Gate、能力保留、授权 nonce 和未知提交语义，不是第二套旁路执行器。

代码清单 7　里程碑验收与结算（参考伪代码）

```
VALIDATE_CONTRIBUTION_DAG(task, milestone, contribution):
    require contribution.header.kind == "contribution"
    require Intact(contribution)
    require contribution.header.networkDomain == task.networkDomain
    require canonicalUnique(contribution.header.parentRoots)
    require immutableContributionBinding(task, milestone, contribution)
    if contribution.header.parentRoots is empty:
        require task.verificationPolicy.allowOfflineContribution
        require contribution.payload.parentClosureRoot == EMPTY_CLOSURE_ROOT
        return []

    gray, black, closure = set(), set(), map()
    DFS(root):
        require root not in gray                // 防环
        if root in black: return
        rec = loadReceiptByRoot(root)
        require rec != null and rec.receiptRoot == root
        require rec.header.kind == "execution" and Intact(rec)
        require rec.header.networkDomain == contribution.header.networkDomain
        require rec.payload.buildContextHash ==
                H(task.networkDomain, task.hash,
                  milestone.id, milestone.claimSlotId)
        require rec.payload.status == "SUCCESS"
        if rec.payload.chainRef exists:
```

```
            require finalitySatisfied(rec.payload.chainRef,
                                      task.settlementPolicy)
        gray.add(root)
        for parent in rec.header.parentRoots: DFS(parent)
        gray.remove(root); black.add(root); closure[root] = rec

    for root in contribution.header.parentRoots: DFS(root)
    require size(closure) <= task.verificationPolicy.maxParentNodes
    require depth(closure) <= task.verificationPolicy.maxParentDepth
    closureRoot = domainSeparatedMerkleRoot(
        "HZB/PoB/PARENT-CLOSURE/v1", sortBytes(keys(closure)))
    require closureRoot == contribution.payload.parentClosureRoot
    return closure

VALID_DISPUTE_CONTROL(payload):
    if payload.state == "OPEN":
        return (payload.authorizesRevalidation is false and
                payload.resolutionRoot is absent)
    return (payload.state in {"UPHELD", "OVERTURNED"} and
            payload.authorizesRevalidation is true and
            payload.resolutionRoot exists)

LOAD_PROTOCOL_ROUND(task, contribution):
    contextHash = H(task.hash, contribution.receiptRoot,
        contribution.payload.parentClosureRoot, claimKey(task, contribution))
    initialId = H("HZB/PoB/round/v1", contextHash,
                  EMPTY_ROOT, EMPTY_ROOT)
    round = createInitialRoundHeadIfAbsent(
        contextHash, RoundPermit(initialId, EMPTY_ROOT, EMPTY_ROOT,
                                 sealedByBPRoot=EMPTY_ROOT))
    require round.roundId == activeRoundHead(contextHash)
    if round.roundId == initialId:
        require round.previousValidationRoot == EMPTY_ROOT
        require round.resolutionRoot == EMPTY_ROOT
    else:
        resolution = loadReceiptByRoot(round.resolutionRoot)
        require resolution.header.kind == "dispute" and Intact(resolution)
        require VALID_DISPUTE_CONTROL(resolution.payload)
        require resolution.payload.pobContextHash == contextHash
        require resolution.payload.state in {"UPHELD", "OVERTURNED"}
        require resolution.payload.targetReceiptRoot ==
                round.previousValidationRoot
        require resolution.payload.authorizesRevalidation is true
        require round.roundId == H("HZB/PoB/round/v1", contextHash,
            round.previousValidationRoot, round.resolutionRoot)
    return round

ADVANCE_PROTOCOL_ROUND(task, contribution, resolutionRoot):
    contextHash = H(task.hash, contribution.receiptRoot,
        contribution.payload.parentClosureRoot, claimKey(task, contribution))
    resolution = loadReceiptByRoot(resolutionRoot)
    require resolution.header.kind == "dispute" and Intact(resolution)
    require VALID_DISPUTE_CONTROL(resolution.payload)
    require resolution.payload.pobContextHash == contextHash
    require resolution.payload.state in {"UPHELD", "OVERTURNED"}
    require resolution.payload.authorizesRevalidation is true
    bp = loadBuildProofByContext(contextHash)
    with serializableTransaction:
        currentBP = lockBuildProof(bp.id)
        current = lockActiveRoundHead(contextHash)
        require current.sealedByBPRoot == EMPTY_ROOT
        previous = loadValidationByRound(contextHash, current.roundId)
        currentDispute = lockDisputeHead(
            currentBP.id, previous.receiptRoot)
```

```
        require currentDispute.receiptRoot == resolution.receiptRoot
        require currentBP.state in {
            "DISPUTED", PENDING_STATE(finalVerdict(previous, currentDispute))}
        require resolution.payload.targetReceiptRoot == previous.receiptRoot
        nextId = H("HZB/PoB/round/v1", contextHash,
                   previous.receiptRoot, resolution.receiptRoot)
        compareAndSwapRoundHead(contextHash, current.roundId,
            RoundPermit(nextId, previous.receiptRoot, resolution.receiptRoot,
                        sealedByBPRoot=EMPTY_ROOT))
    return nextId

requireValidationBinding(rec, validationKey, contextHash,
                         contribution, task, roundPermit):
    require rec.header.kind == "validation" and Intact(rec)
    require rec.header.networkDomain == task.networkDomain
    require rec.header.subjectHash == validationKey
    require rec.payload.pobContextHash == contextHash
    require rec.payload.contributionReceiptRoot == contribution.receiptRoot
    require rec.payload.verificationPolicyHash == task.verificationPolicy.hash
    require rec.payload.validationRoundId == roundPermit.roundId
    require rec.payload.claimSetRoot ==
            root(requiredClaims(task.verificationPolicy))

GET_VALIDATION_ONCE(task, contribution, closure, roundPermit):
    contextHash = H(task.hash, contribution.receiptRoot,
        contribution.payload.parentClosureRoot, claimKey(task, contribution))
    require roundPermit.roundId == activeRoundHead(contextHash)
    validationKey = H("HZB/PoB/validation/v1", contextHash,
        task.verificationPolicy.hash, roundPermit.roundId)
    existing = loadValidationByKey(validationKey)
    if existing exists:
        requireValidationBinding(existing, validationKey, contextHash,
                                 contribution, task, roundPermit)
        return existing
    with keyedLease(validationKey):
        existing = loadValidationByKey(validationKey)
        if existing exists:
            requireValidationBinding(existing, validationKey, contextHash,
                                     contribution, task, roundPermit)
            return existing
        reports = verifyRequiredClaims(contextHash, task.verificationPolicy)
        require signedReports(reports)
        require independentQuorum(reports, task.verificationPolicy)
        require noConflictOfInterest(reports)
        receipt = makeSignedValidationReceipt(
            subjectHash=validationKey,
            parentRoots=[contribution.receiptRoot],
            pobContextHash=contextHash,
            contributionReceiptRoot=contribution.receiptRoot,
            verificationPolicyHash=task.verificationPolicy.hash,
            validationRoundId=roundPermit.roundId,
            claimSetRoot=root(requiredClaims(task.verificationPolicy)),
            reportSetRoot=root(reports),
            verifierSetHash=root(verifierSubjects(reports)),
            verdict=deterministicVerdict(reports),
            appealDeadline=task.appealDeadline)
        require roundPermit.roundId == activeRoundHead(contextHash)
        insertValidationUnique(validationKey, receipt)
        return receipt

PENDING_STATE(verdict):
    return ("ACCEPTED_PENDING_APPEAL" if verdict == "ACCEPT"
            else "REJECTED_PENDING_APPEAL")
```

```
APPLY_DISPUTE_RECEIPT_ONCE(bpId, disputeReceipt, now):
    require disputeReceipt.header.kind == "dispute" and Intact(disputeReceipt)
    require VALID_DISPUTE_CONTROL(disputeReceipt.payload)
    with serializableTransaction:
        currentBP = lockBuildProof(bpId)
        currentHead = lockDisputeHead(
            bpId, disputeReceipt.payload.targetReceiptRoot)
        require disputeReceipt.payload.pobContextHash == currentBP.pobContextHash
        require disputeReceipt.payload.targetReceiptRoot ==
                currentBP.validationReceiptRoot
        if disputeReceipt.payload.state == "OPEN":
            require currentBP.state in {
                "ACCEPTED_PENDING_APPEAL", "REJECTED_PENDING_APPEAL"}
            require now <= currentBP.appealDeadline
        else:
            require currentBP.state == "DISPUTED"
            require currentHead.payload.state == "OPEN"
        writeDisputeHeadAndBPStateAtomically(
            currentBP.id, currentHead, disputeReceipt,
            expectedBPVersion=currentBP.stateVersion,
            targetState="DISPUTED")

SYNC_BP_PROTOCOL_STATE(bp, validation):
    with serializableTransaction:
        currentBP = lockBuildProof(bp.id)
        currentRound = lockActiveRoundHead(validation.payload.pobContextHash)
        currentDispute = lockDisputeHead(
            currentBP.id, validation.receiptRoot)
        require validation.payload.validationRoundId == currentRound.roundId
        require currentRound.roundId ==
                activeRoundHead(validation.payload.pobContextHash)
        if currentBP.state in {"FINAL_ACCEPTED", "FINAL_REJECTED"}:
            return currentBP, currentDispute
        if currentBP.validationReceiptRoot != validation.receiptRoot:
            require currentRound.previousValidationRoot ==
                    currentBP.validationReceiptRoot
            previousValidation = loadReceiptByRoot(
                currentRound.previousValidationRoot)
            resolution = loadReceiptByRoot(currentRound.resolutionRoot)
            require resolution.header.kind == "dispute" and Intact(resolution)
            require VALID_DISPUTE_CONTROL(resolution.payload)
            require resolution.payload.pobContextHash ==
                    validation.payload.pobContextHash
            require resolution.payload.targetReceiptRoot ==
                    previousValidation.receiptRoot
            require currentBP.state in {
                "DISPUTED",
                PENDING_STATE(finalVerdict(previousValidation, resolution))}
            currentBP = compareAndSwapBuildProofState(
                currentBP.id, currentBP.stateVersion, "VALIDATING",
                validationReceiptRoot=validation.receiptRoot)
        if currentBP.state == "DRAFT":
            currentBP = compareAndSwapBuildProofState(
                currentBP.id, currentBP.stateVersion, "VALIDATING",
                validationReceiptRoot=validation.receiptRoot)
        if currentBP.state == "VALIDATING":
            currentBP = compareAndSwapBuildProofState(
                currentBP.id, currentBP.stateVersion,
                PENDING_STATE(validation.payload.verdict),
                validationReceiptRoot=validation.receiptRoot)
        if currentBP.state == "DISPUTED" and currentDispute is absent:
            currentBP = compareAndSwapBuildProofState(
                currentBP.id, currentBP.stateVersion,
                PENDING_STATE(validation.payload.verdict),
```

```
                validationReceiptRoot=validation.receiptRoot)
        if currentDispute exists:
            require VALID_DISPUTE_CONTROL(currentDispute.payload)
            require currentDispute.payload.targetReceiptRoot ==
                    validation.receiptRoot
            if currentDispute.payload.state == "OPEN":
                require currentBP.state == "DISPUTED"
                return currentBP, currentDispute
            require currentBP.state == "DISPUTED"
            currentBP = compareAndSwapBuildProofState(
                currentBP.id, currentBP.stateVersion,
                PENDING_STATE(finalVerdict(validation, currentDispute)),
                validationReceiptRoot=validation.receiptRoot)
        require currentBP.state ==
                PENDING_STATE(finalVerdict(validation, currentDispute))
        return currentBP, currentDispute

ATOMIC_FINAL_ACCEPTED(task, contribution, validation, bp, dispute,
                      expectedClaimKey, claimNullifier, expectedBPVersion):
    with serializableTransaction:
        currentBP = lockBuildProof(bp.id)
        currentRound = lockActiveRoundHead(validation.payload.pobContextHash)
        currentDispute = lockDisputeHead(
            currentBP.id, validation.receiptRoot)
        require currentBP.state == "ACCEPTED_PENDING_APPEAL"
        require currentBP.stateVersion == expectedBPVersion
        require expectedClaimKey == claimKey(task, contribution)
        require claimNullifier == contribution.payload.claimNullifier
        require validation.payload.validationRoundId == currentRound.roundId
        require currentRound.roundId ==
                activeRoundHead(validation.payload.pobContextHash)
        require root(currentDispute) == root(dispute)
        require appealClosed(validation, currentDispute)
        require not isOpen(currentDispute)
        require finalVerdict(validation, currentDispute) == "ACCEPT"
        candidate = freezeBuildProof(
            currentBP, state="FINAL_ACCEPTED",
            disputeRoot=root(currentDispute))
        candidate.bpRoot = H("HZB/PoB/BP/v1", candidate.networkDomain,
            canonicalEncode(candidate.without("bpRoot")))
        require currentRound.sealedByBPRoot in {
            EMPTY_ROOT, candidate.bpRoot}
        prospectiveRegistry = previewUniqueBindings(
            expectedClaimKey, claimNullifier, candidate.bpRoot)
        require PoBValid(task, contribution, validation, candidate,
                         prospectiveRegistry)
        sealRoundHeadIfUnsealed(
            currentRound, candidate.bpRoot)
        bindUniqueOrSame("claim_registry", expectedClaimKey, candidate.bpRoot)
        bindUniqueOrSame("contribution_nullifier", claimNullifier,
                         candidate.bpRoot)
        insertBuildProofUnique(candidate.bpRoot, candidate)
        compareAndSwapBuildProofState(
            candidate.id, expectedBPVersion, "FINAL_ACCEPTED")
    return candidate

ATOMIC_FINAL_REJECTED(task, validation, bp, dispute, expectedBPVersion):
    with serializableTransaction:
        currentBP = lockBuildProof(bp.id)
        currentRound = lockActiveRoundHead(validation.payload.pobContextHash)
        currentDispute = lockDisputeHead(
            currentBP.id, validation.receiptRoot)
        require currentBP.state == "REJECTED_PENDING_APPEAL"
        require currentBP.stateVersion == expectedBPVersion
```

```
        require validation.payload.validationRoundId == currentRound.roundId
        require currentRound.roundId ==
                activeRoundHead(validation.payload.pobContextHash)
        require root(currentDispute) == root(dispute)
        require appealClosed(validation, currentDispute)
        require not isOpen(currentDispute)
        require finalVerdict(validation, currentDispute) == "REJECT"
        candidate = freezeBuildProof(
            currentBP, state="FINAL_REJECTED",
            disputeRoot=root(currentDispute))
        candidate.bpRoot = H("HZB/PoB/BP/v1", candidate.networkDomain,
            canonicalEncode(candidate.without("bpRoot")))
        require currentRound.sealedByBPRoot in {
            EMPTY_ROOT, candidate.bpRoot}
        require rejectedTerminalBinding(
            task, validation, candidate, currentDispute)
        sealRoundHeadIfUnsealed(
            currentRound, candidate.bpRoot)
        insertBuildProofUnique(candidate.bpRoot, candidate)
        compareAndSwapBuildProofState(
            candidate.id, expectedBPVersion, "FINAL_REJECTED")
    return candidate

FINALIZE_MILESTONE(sourceApp, buildTaskId, milestoneId,
                   contributionRoot):
    task = loadSignedFrozenBuildTask(buildTaskId)
    require validPublisherSignature(task.publisherSignature)
    require task.sourceApp == sourceApp and signedOrigin(sourceApp)
    milestone = loadFrozenMilestone(task, milestoneId)
    contribution = loadReceiptByRoot(contributionRoot)
    closure = VALIDATE_CONTRIBUTION_DAG(task, milestone, contribution)
    roundPermit = LOAD_PROTOCOL_ROUND(task, contribution)
    validation = GET_VALIDATION_ONCE(
        task, contribution, closure, roundPermit)
    bp = loadOrCreateBPDraft(task, contribution, validation)
    if bp.state in {"FINAL_ACCEPTED", "FINAL_REJECTED"}:
        return PREPARE_SETTLEMENT_ONCE(task, bp, validation)
    bp, dispute = SYNC_BP_PROTOCOL_STATE(bp, validation)
    if bp.state == "DISPUTED": return Hold("DISPUTED")
    if now < validation.payload.appealDeadline:
        return Hold("APPEAL_WINDOW_OPEN")
    resolved = finalVerdict(validation, dispute)
    if resolved == "ACCEPT":
        bp = ATOMIC_FINAL_ACCEPTED(task, contribution, validation, bp, dispute,
            claimKey(task, contribution), contribution.payload.claimNullifier,
            bp.stateVersion)
    else:
        bp = ATOMIC_FINAL_REJECTED(
            task, validation, bp, dispute, bp.stateVersion)
    return PREPARE_SETTLEMENT_ONCE(task, bp, validation)

NONTERMINAL_SETTLEMENT_STATES = {
    "READY_TO_SUBMIT", "SUBMITTING", "SUBMITTED", "UNKNOWN"}

PREPARE_SETTLEMENT_ONCE(task, bp, validation):
    require bp.state in {"FINAL_ACCEPTED", "FINAL_REJECTED"}
    obligation = loadFrozenObligation(task, bp.milestoneId)
    require finalTerminalBinding(task, bp, validation, obligation)
    if bp.state == "FINAL_ACCEPTED":
        contribution = loadReceiptByRoot(bp.contributionReceiptRoot)
        uniqueRegistry = loadUniqueRegistrySnapshot(bp.bpRoot)
        require PoBValid(task, contribution, validation, bp,
                         uniqueRegistry)
    else:
```

```
    require RejectBound(task, validation, bp, obligation)
branch = "RELEASE" if bp.state == "FINAL_ACCEPTED" else "REFUND"
settlementKey = H(
    "HZB/SETTLEMENT/v1", networkDomain, task.buildTaskId,
    task.version, bp.milestoneId, obligation.id, obligation.version)
call = frozenBranchCall(obligation, branch, bp)
require call.chainNonce == obligation.effectPlan.chainNonce
require call.feePolicyHash == obligation.effectPlan.feePolicyHash
require H(canonicalActionProjection(networkDomain, call)) == call.callHash
existing = loadSettlement(settlementKey)
if existing exists:
    requireFrozenSettlementBinding(
        existing, task.versionHash, bp.bpRoot, validation.receiptRoot,
        branch, call.callHash, obligation.id, obligation.version)
    if existing is final: return existing.receiptRoot
    require existing.state in NONTERMINAL_SETTLEMENT_STATES
    return Pending(settlementKey)

simulation = simulateFixedState(call, task.settlementPolicy)
verification = verifySettlementCall(call, simulation, task.settlementPolicy)
gate = policyGate(gateInput(task, call, verification, validation), now)
require gate.effect == "ALLOW"
require hashCanonical(task.settlementPolicy.finalityPolicy) ==
        task.finalityPolicyHash
authorizationMessage = typedAuthorizationMessage(
    domain="HZB/AUTH/v1", networkDomain=networkDomain,
    taskVersionHash=task.versionHash, planHash=task.planHash,
    principal=task.principal, actionHash=call.callHash,
    decisionHash=gate.decisionHash,
    capabilityId=call.capabilityId,
    capabilityHash=call.capabilityHash,
    capabilityVersion=call.capabilityVersion,
    revocationVersion=call.revocationVersion,
    authorizationNonce=call.authorizationNonce,
    expiresAt=call.authorizationExpiresAt)
authorization = loadTypedAuthorization(authorizationMessage)
authorizationHash = hashCanonical(authorization)
require verifyTypedAuthorizationMessage(
    authorization, authorizationMessage, now)
submitPayload = frozenSubmitPayload(call)
frozenSubmitPayloadHash = hashFrozenSubmitPayload(submitPayload)

with serializableTransaction:
    settlement = lockSettlement(settlementKey)
    if settlement exists:
        requireFrozenSettlementBinding(
            settlement, task.versionHash, bp.bpRoot,
            validation.receiptRoot, branch, call.callHash,
            obligation.id, obligation.version)
        if settlement is final: return settlement.receiptRoot
        require settlement.state in NONTERMINAL_SETTLEMENT_STATES
        return Pending(settlementKey)
    lockAndRequireOutstanding(
        obligation.id, obligation.version, call.amount)
    capability = lockCapability(call.capabilityId)
    require capability.version == call.capabilityVersion
    require capability.revocationVersion == call.revocationVersion
    require capability.hash == call.capabilityHash
    require capabilityAllows(capability, call)
    assert_permit_decision(
        gate, task.versionHash, call.callHash, now)
    assert_typed_authorization(
        authorization, expectedPrincipal=task.principal,
        networkDomain=networkDomain,
```

```
            taskVersionHash=task.versionHash, planHash=task.planHash,
            actionHash=call.callHash, decisionHash=gate.decisionHash,
            capabilityId=call.capabilityId,
            capabilityHash=call.capabilityHash,
            capabilityVersion=call.capabilityVersion,
            revocationVersion=call.revocationVersion,
            authorizationNonce=call.authorizationNonce, now=now)
        consume_approval_nonces_once(
            task.versionHash, gate.approvalBindings,
            expected_hash=gate.approvalSetHash, now=now)
        quotaReservation = reserveQuota(
            capability.id, call.quotaDelta)
        consumeAuthorizationNonceOnce(
            capability.id, call.authorizationNonce)
        chainNonceReservation = reserveChainNonceOnce(
            call.chainId, call.sender, call.chainNonce)
        obligationReservation = reserveObligation(
            settlementKey, obligation.id)
        attemptNo = reserveNextAttemptNo(settlementKey)
        activeAttemptId = H(settlementKey, attemptNo,
                            call.callHash, call.chainNonce)
        upsertSettlement(
            settlementKey=settlementKey,
            taskVersionHash=task.versionHash,
            buildProofRoot=bp.bpRoot,
            validationReceiptRoot=validation.receiptRoot,
            branch=branch, callHash=call.callHash,
            authorizationHash=authorizationHash,
            decisionHash=gate.decisionHash,
            obligationId=obligation.id,
            obligationVersion=obligation.version,
            capabilityId=capability.id,
            quotaReservationId=quotaReservation.id,
            chainId=call.chainId, chainDomain=call.chainDomain,
            sender=call.sender, chainNonce=call.chainNonce,
            chainNonceReservationId=chainNonceReservation.id,
            obligationReservationId=obligationReservation.id,
            attemptNo=attemptNo, activeAttemptId=activeAttemptId,
            frozenSubmitPayload=submitPayload,
            frozenSubmitPayloadHash=frozenSubmitPayloadHash,
            expectedStateDeltaRoot=simulation.expectedStateDeltaRoot,
            frozenFinalityPolicy=task.settlementPolicy.finalityPolicy,
            frozenFinalityPolicyHash=task.finalityPolicyHash,
            recipient=call.recipient, asset=call.asset,
            amount=call.amount, rowVersion=1,
            state="READY_TO_SUBMIT")
        outbox.putIfAbsent(activeAttemptId, "SubmitSettlement", {
            settlementKey, attemptId: activeAttemptId,
            chainNonce: call.chainNonce,
            frozenSubmitPayload: submitPayload })
    return Pending(settlementKey)

OBSERVE_AFTER_CAS_CONFLICT(settlementKey, attemptId, callHash):
    current = loadSettlement(settlementKey)
    if current is final: return current.receiptRoot
    require current.activeAttemptId == attemptId
    require current.callHash == callHash
    return Pending(settlementKey)

CAS_SETTLEMENT_OR_OBSERVE(settlementKey, expectedRowVersion,
                          allowedFrom, target, attemptId,
                          callHash, evidence):
    try:
        return CASSettlementState(
```

```
            settlementKey, expectedRowVersion, allowedFrom, target,
            attemptId=attemptId, callHash=callHash, evidence=evidence)
    on CASConflict:
        return OBSERVE_AFTER_CAS_CONFLICT(
            settlementKey, attemptId, callHash)

SUBMIT_OUTBOX(event):
    row = lockSettlement(event.settlementKey)
    if row is final: return row.receiptRoot
    require row.state in {"READY_TO_SUBMIT", "SUBMITTING", "UNKNOWN"}
    require event.attemptId == row.activeAttemptId
    require hashFrozenSubmitPayload(event.frozenSubmitPayload) ==
            row.frozenSubmitPayloadHash
    require event.chainNonce == row.chainNonce
    rawTx = row.rawTx or deterministicSign(event.frozenSubmitPayload)
    txHash = hashRawTransaction(rawTx)
    try:
        row = atomicallyStoreBeforeIO(
            settlementKey=row.settlementKey,
            expectedRowVersion=row.rowVersion,
            allowedFrom={"READY_TO_SUBMIT", "SUBMITTING", "UNKNOWN"},
            target="SUBMITTING", attemptId=event.attemptId,
            callHash=row.callHash, rawTxHash=txHash,
            rawTx=encrypted(rawTx))
        expectedRowVersion = row.rowVersion
        result = broadcastExact(rawTx)
        if result.accepted or result.alreadyKnown:
            return CAS_SETTLEMENT_OR_OBSERVE(
                row.settlementKey, expectedRowVersion, {"SUBMITTING"},
                "SUBMITTED", event.attemptId, row.callHash, result)
        return CAS_SETTLEMENT_OR_OBSERVE(
            row.settlementKey, expectedRowVersion, {"SUBMITTING"},
            "UNKNOWN", event.attemptId, row.callHash, result)
    on CASConflict:
        return OBSERVE_AFTER_CAS_CONFLICT(
            row.settlementKey, event.attemptId, row.callHash)
    on ProvenNoBytesSent:
        return CAS_SETTLEMENT_OR_OBSERVE(
            row.settlementKey, expectedRowVersion, {"SUBMITTING"},
            "READY_TO_SUBMIT", event.attemptId, row.callHash,
            {txHash, provenNoBytesSent: true})
    on AnyPossiblySubmittedError:
        return CAS_SETTLEMENT_OR_OBSERVE(
            row.settlementKey, expectedRowVersion, {"SUBMITTING"},
            "UNKNOWN", event.attemptId, row.callHash,
            {txHash, submissionUncertain: true})

RECONCILE_UNCERTAIN(settlementKey):
    row = loadSettlement(settlementKey)
    if row is final: return row.receiptRoot
    require row.state in {"SUBMITTING", "UNKNOWN"}
    expectedRowVersion = row.rowVersion
    evidence = queryByTxHashAndSenderNonce(
        row.rawTxHash, row.chainId, row.sender, row.chainNonce)
    if evidence.sameTransactionFound:
        return CAS_SETTLEMENT_OR_OBSERVE(
            settlementKey, expectedRowVersion, {"SUBMITTING", "UNKNOWN"},
            "SUBMITTED", row.activeAttemptId, row.callHash, evidence)
    else if evidence.differentTransactionAtNonce:
        return CAS_SETTLEMENT_OR_OBSERVE(
            settlementKey, expectedRowVersion, {"SUBMITTING", "UNKNOWN"},
            "NONCE_CONFLICT", row.activeAttemptId, row.callHash, evidence)
    else:
        return CAS_SETTLEMENT_OR_OBSERVE(
```

```
            settlementKey, expectedRowVersion, {"SUBMITTING", "UNKNOWN"},
            "UNKNOWN", row.activeAttemptId, row.callHash, evidence)

FINALIZE_SETTLEMENT(settlementKey, finalityEvidence):
    row = lockSettlement(settlementKey)
    if row is final: return loadReceipt(row.receiptRoot)
    require row.state in {"SUBMITTING", "SUBMITTED", "UNKNOWN"}
    require hashCanonical(row.frozenFinalityPolicy) ==
            row.frozenFinalityPolicyHash
    require finalityPolicySatisfied(
        finalityEvidence, row.frozenFinalityPolicy)
    require finalityEvidence.txHash == row.rawTxHash
    require finalityEvidence.callHash == row.callHash
    require finalityEvidence.executionStatus == "SUCCESS"
    require finalityEvidence.observedStateDeltaRoot == row.expectedStateDeltaRoot
    require transactionEnvelopeMatches(row.rawTx, finalityEvidence)
    executionReceipt = ensureFinalExecutionReceipt(row, finalityEvidence)
    settlementReceipt = makeSignedReceipt(
        kind="settlement", subjectHash=settlementKey,
        parentRoots=[row.validationReceiptRoot,
                     executionReceipt.receiptRoot],
        payload=frozenSettlementPayload(row, finalityEvidence))
    with serializableTransaction:
        row = lockSettlement(settlementKey)
        if row is final:
            require row.receiptRoot == settlementReceipt.receiptRoot
            return loadReceipt(row.receiptRoot)
        require row.state in {"SUBMITTING", "SUBMITTED", "UNKNOWN"}
        require row.activeAttemptId == finalityEvidence.attemptId
        require row.rawTxHash == finalityEvidence.txHash
        require row.callHash == finalityEvidence.callHash
        require hashCanonical(row.frozenFinalityPolicy) ==
                row.frozenFinalityPolicyHash
        require finalityPolicySatisfied(
            finalityEvidence, row.frozenFinalityPolicy)
        require finalityEvidence.executionStatus == "SUCCESS"
        require finalityEvidence.observedStateDeltaRoot == row.expectedStateDeltaRoot
        require transactionEnvelopeMatches(row.rawTx, finalityEvidence)
        require obligationStillReservedBy(
            row.obligationReservationId, row.obligationId, settlementKey)
        clearOutstandingObligation(
            row.obligationId, row.obligationReservationId)
        commitCapabilityQuota(row.quotaReservationId)
        commitChainNonce(row.chainNonceReservationId)
        markSettlementFinal(row,
            "FINAL_RELEASED" if row.branch == "RELEASE"
                             else "FINAL_REFUNDED",
            settlementReceipt.receiptRoot)
        insertReceiptUnique(settlementKey, settlementReceipt)
        outbox.putIfAbsent(H(settlementKey, "FINAL"),
            "SettlementFinalized", settlementReceipt.receiptRoot)
    return settlementReceipt
```

validation receipt 以唯一 validationKey 创建；新轮仅由绑定前一 validation 与 resolution 根的闭合争议授权。父 DAG 校验环、缺根、跨里程碑、状态、最终性和规模。BP 唯一键、终态与 active round 封存同事务提交。Release/Refund 共用 settlementKey；SUBMITTING/UNKNOWN 只重播已保存 rawTx。Outbox 以稳定 eventId 去重。

实现层须对 validationKey、claimKey、claimNullifier、settlementKey、authorizationNonce、chainNonce、receipt_by_settlement、outbox eventId 建唯一约束。最终清偿、settlement receipt 与 final outbox 在同一串行化事务提交，消除“义务已清但回执丢失”的崩溃窗口。

7 安全架构与威胁模型

7.1 安全目标、资产与攻击者

HIVM 的安全目标不是证明“大模型永不出错”，而是让不确定推理始终停留在可观察、可撤销的边界内。即使用户输入、检索材料、模型或外部工具被操纵，单点失陷也不应自然升级为跨租户取数、任意联网、任意签名或无上限资产损失。系统追求四项可检验性质：未授权副作用不可达；授权参数与执行参数不可静默分离；每次关键状态变化可关联到同版本意图、证据、策略和主体；失败范围与补偿责任可以被明确界定。它们是纵深防御目标，不是“零风险”承诺。

受保护资产分为四类。价值与授权资产包括钱包密钥、多签角色、额度、托管义务与结算键；执行完整性资产包括意图、任务图、策略版本、交易字节、nonce 和回执；数据与隐私资产包括租户文档、交付物、来源与知识产权材料；治理与共建资产包括 PoB 任务、验证决定、争议、声誉、Venture Cell 角色和暂停开关。资产分类直接决定保留、路由与披露范围。

至少存在六条信任边界：不可信内容进入认知区；概率性模型输出进入确定性工具协议；工具沙箱访问文件、网络和 API；执行控制跨入钱包/KMS 与结算区；租户命名空间进入共享缓存、索引和日志；控制面把模型、策略、容器和适配器发布到数据面。跨越边界的对象都必须有模式、身份、完整性标签、租户标签和生命周期，不能把“看起来像系统命令”的文本视为授权。

攻击者模型涵盖能提交任意提示、文件、URL 和参数的外部用户，能够污染知识库的内容发布者，恶意或失陷的租户、模型、工具、RPC 与预言机供应商，取得部分 CI/CD 或运维权限的内部人，观察公共内存池并实施抢先/夹击的搜索者，以及控制部分验证源或创建女巫身份的经济攻击者。基础分析不假设合格密码算法被攻破，也不假设所有独立故障域同时失陷；这些更强情形应作为灾难恢复演练，而非被设计文档悄然排除。

7.2 三安全分区

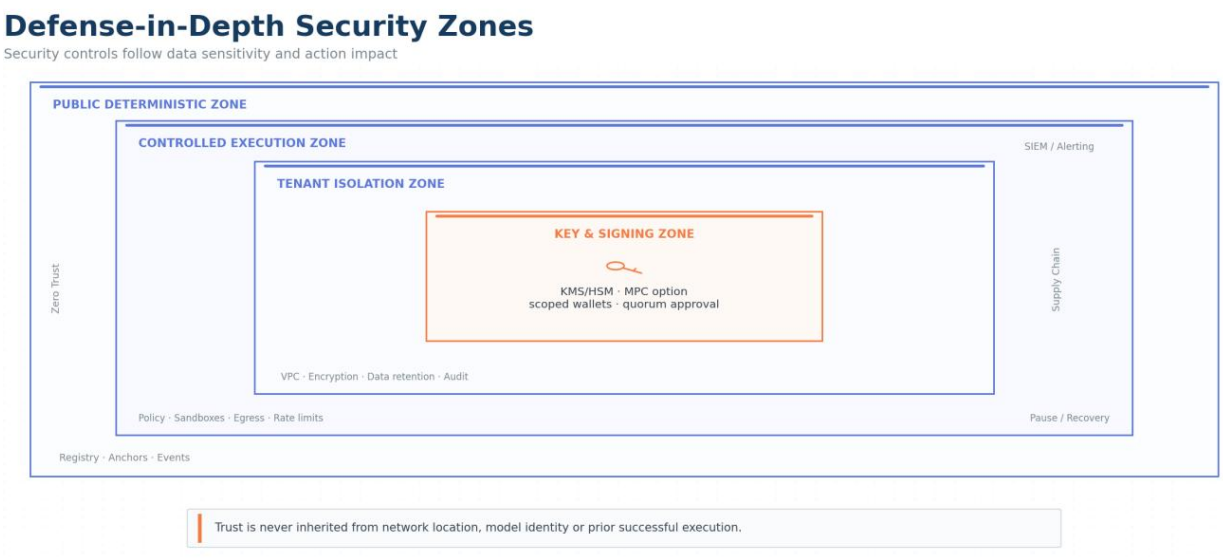

图 5 HIVM 三安全分区与信任边界

不可信认知区运行提示编排、RAG 和模型推理，默认输出可能含幻觉、注入或投毒。系统指令、用户内容、检索片段和工具返回分别封装并保留来源，外部内容不能提升为系统权威。知识入库执行来源和许可检查、版本/时间记录、恶意内容扫描、租户命名空间隔离与可撤销索引；高风险事实使用独立来源复核。模型进程没有生产密钥、宿主文件系统或直接互联网访问，缓存键必须包含租户与敏感级别。OWASP 与 MITRE 的公开知识库将提示注入、敏感信息泄露、供应链、工具滥用、目标劫持和级联故障列为重要 AI/Agent 风险[14-16]，但分类清单只能帮助覆盖场景，不能代替系统边界。

HHA 与其他外部应用一样是不可信入口：适配器必须验证来源签名、版本、租户、用户同意与数据级别。Agent 可预检交付，但不能成为高价值 PoB 的唯一验证者或法律责任主体。

执行控制区是模型外部的强制策略点。模型只能提交符合模式的计划；策略引擎依据签名能力检查工具、动作、目标域/合约、参数、数据等级、次数、金额、费用和有效期。工具在短生命周期沙箱运行，采用只读根文件系统、最小挂载、系统调用过滤以及 CPU、内存、时长和并发预算。网络出口代理默认拒绝，解析后仍需阻断环回、链路本地、云元数据和私网地址；重定向与 DNS 重绑定应再次检查。模型输出进入 SQL、Shell、模板、HTTP 或 ABI 前必须按目标语法重新编码，不能依赖模型“遵守提示”。

状态改变前在固定 chainId、区块和合约代码摘要上模拟，检查余额差分、授权额度、滑点、接收方、未知外部调用和失败路径。新接收方、首次工具、跨租户导出、高金额、无限授权、

代理升级、策略降级和模拟差异应触发人工或多方审批。审批界面展示解码后的真实调用、最坏损失、费用上限和版本摘要，而不是只展示模型生成的友好说明。

密钥与结算区不接受自然语言，只接受字段完整、未过期且带策略决定的规范化授权对象。长期根密钥、服务签名密钥、用户钱包、会话密钥和恢复份额按用途与故障域分离；生产密钥宜置于 KMS/HSM 不可导出槽，服务只获得特定算法、地址和策略下的签名权。限权钱包设置单笔/日累计上限、资产和合约白名单、禁止无限授权、会话过期和紧急撤权。大额转移、升级及暂停恢复使用职责分离的多签与时间锁；密钥生成、分发、存储、轮换、撤销、归档和销毁应纳入组织级生命周期管理[33]。密码模块合规或有效签名只说明部分控制成立，并不证明审批者理解交易或业务决策正确。

7.3 控制链与安全不变量

HIVM 把防护组织为“输入隔离—能力缩减—策略判定—沙箱执行—固定状态模拟—绑定授权—限权签名—最终性观察—回执复核”的控制链。任一环节通过都不自动授权下一环节。尤其是，模型安全过滤不能取代能力控制，模拟成功不能取代授权，签名有效不能取代链域与时效检查，交易确认也不能取代链下交付验证。

设计与测试至少应共同检查以下不变量：

1. 模型、RAG 与普通工具进程永不持有生产密钥、签名份额或可绕过策略点的通用凭据。
2. 未列入能力清单的工具、动作、目标和网络出口默认拒绝；授权不能由模型生成字段自行扩大。
3. 每个写动作绑定租户、意图版本、计划摘要、chainId、合约、nonce、deadline、资产、数量、接收方和费用上限，缺一失败关闭。
4. 每个 effectKey 至多产生一次副作用，且幂等保留与累计额度消费原子完成。
5. 高风险动作同时通过固定状态模拟、确定性策略与独立批准；广播前差异越过阈值即重新验证和授权。
6. 跨租户隔离覆盖数据库、向量库、缓存、日志、临时介质和批处理，不因共享模型而消失。
7. 从意图到最终回执的关联链不得断裂；RPC 分歧、版本漂移或证据过期不得报告完整成功。

8. 模型、工具、策略、镜像和合约适配器必须匹配批准摘要；变更分权、分阶段发布并保留兼容回退。
9. 受限数据和秘密不得进入普通提示历史、错误栈或第三方遥测，审计字段遵循最小披露。
10. 发现密钥疑似泄漏、异常资金流、越权调用或严重共识源分歧时，自动化让位于撤销、限流、暂停和人工处置。
11. PoB 未达最终验收或申诉窗未关闭时不得结算；同一建设记录不得在不同里程碑重复申领。
12. Venture Cell 成员资格不自动赋权，每个任务、贡献、验证与结算均绑定具体版本与能力。

这些不变量应转化为属性测试、状态机模型检查、负向隔离测试和故障注入断言。若实现为了提高“任务成功率”而放宽其中任一项，指标上升可能只是安全拒绝被错误计为失败并被消除。

7.4 主要威胁与控制矩阵

威胁识别以身份仿冒、篡改、抵赖、信息泄露、拒绝服务和权限提升为骨架，并结合 Agent 特有的提示/目标劫持、记忆投毒、过度代理权和级联故障。表 6 为设计期优先级，剩余风险列说明控制为何不能被表述为绝对安全。

表 6　HIVM 主要威胁、规划控制与剩余风险

威胁与路径	主要影响	规划控制	剩余风险
直接/间接提示注入	越权规划、秘密外传	内容/指令分离、外部策略、最小能力	语义检测可绕过，故模型永不直接获权
RAG 污染、旧索引占据召回	错误事实、持久后门	来源/版本/时效、撤销索引、多源复核	可信来源可能同源出错
工具越权与不安全输出	RCE、删库、转账	类型化协议、逐参数授权、沙箱、预算	允许工具自身仍可能有漏洞
SSRF、重定向与 DNS 重绑定	凭据窃取、内网探测	独立出口代理、解析后 IP 检查、默认拒绝	合法白名单服务可能被攻陷
模型/工具/策略供应链替换	后门、行为漂移、门禁旁路	摘要锁定、SLSA 来源、签名、双人发布[32]	来源证明不等于源码无恶意
恶意 RPC、预言机或索引器	错误模拟、报价与确认	不同故障域多源、区块绑定、偏差阈值	串谋或共同上游可一致欺骗

威胁与路径	主要影响	规划控制	剩余风险
旧授权、跨链或重复提交重放	重复付款	EIP-712 域、nonce、deadline、单次消费[26-28]	应用遗漏字段仍会形成漏洞
前置交易、夹击与排序操纵	滑点和费用损失	紧滑点、再模拟、私有通道、费用上限	MEV 与区块构建者风险不能根除
密钥泄漏或签名钓鱼	资产和控制面失陷	HSM/KMS、限权钱包、多签、清晰解码	合法签名不证明知情同意
跨租户缓存/索引/日志泄漏	隐私和商业数据泄漏	对象租户标签、分钥、隔离负测	共享模型存在记忆性泄漏可能
回执伪造或工具谎报成功	错误记账、抵赖	请求—计划—交易关联、多 RPC、签名回执	链上事件不证明链下交付
伪造/抄袭/重复 PoB 证据	错误验收与重复奖励	工件、来源、里程碑和父回执绑定	原创性和 IP 权属仍需专业/法律审查
女巫验证者、串谋与利益冲突	虚假通过、申诉捕获	故障域法定人数、COI 披露、抽样与上诉	链下关系难完全观测
HHA 适配器伪冒或跨租户越界	伪造需求、数据泄露	应用域签名、版本、同意和租户绑定	入口运营方密钥可被攻破
里程碑提前或重复释放	超额支付、托管失衡	验收/申诉门、唯一结算键、余额对账	托管偿付与目标链最终性风险
声誉成为隐性授权	新人歧视、权限绕过	声誉仅路由/风险输入，不替代能力与证据	反馈循环仍可形成偏差
递归 Agent、证明或 RPC 扇出	成本失控、拒绝服务	分层预算、深度限制、背压和熔断	分布式低速耗尽仍需容量治理

7.5 证据边界、隐私与共同失效

哈希证明“当前字节与先前承诺一致”，不能证明数据来自现实；来源签名证明特定密钥签过，不能证明签发者没有误报；TEE 报告在硬件根、补丁和验证策略假设下绑定软件度量，不能证明程序无逻辑错误或侧信道；zkML 证明电路中的关系，不能证明输入真实、模型公平或电路外预处理正确；多源一致只在来源足够独立时降低单点故障。将这些证据组合时，必须绑定同一 taskId、planHash、inputRoot、chainId 和时间窗口，否则几个分别有效的证明可能属于不同任务。

审计与隐私存在真实张力。完整保存提示、检索片段和工具响应便于复盘，却可能暴露个人数据、商业秘密和攻击载荷。HIVM 因而采用三层披露：公共层仅含任务标识、算法、时间和承

诺；授权审计层含脱敏元数据与包含证明；受限层保存必要原文并使用独立密钥、保留期与访问日志。受限原文到期删除后，链上承诺仍能说明曾提交某内容，但不再能恢复内容，回执应同步标记复核能力下降。回执不是保存模型隐式思维链的理由。

7.6 供应链、可观测性与事件响应

模型权重、量化参数、提示模板、工具包、容器、策略、ABI 和 Chain Adapter 均是发布物。构建流水线记录源提交、依赖锁、构建器身份、SBOM、测试、签名和不可变摘要；部署入口拒绝未知、撤销或降级制品。SLSA 可说明制品如何构建及其来源[32]，但不能替代源码审查、模型评测和运行时监控。策略升级先在影子模式重放脱敏任务，比较新旧决定；若指令语义或规范哈希变化，应提升协议版本并保留旧回执验证器。

可观测性围绕任务因果关系组织。每个 taskId 下记录排队、执行、验证、等候审批、广播和最终确认的跨度，以及编译拒绝率、验证冲突率、策略升级率、重复投递消除数、未决广播时长、补偿数和待锚定积压。遥测与密码回执互补：前者支持运行诊断，后者支持内容完整性；两者都不应收集私钥、完整受限提示或未公开交易参数。

事件响应先撤销能力、限流并隔离沙箱，同时暂停相关 PoB 验证与结算但保留争议证据；再轮换密钥、切换故障域、回退到经批准且状态兼容的版本，最后冻结日志、镜像、证据和链上高度用于取证。Pausable 与时间锁可作参考[30-31]，但必须防止单一管理员滥用。链上已确认交易通常不能“回滚”，版本回退、任务取消和资产补偿是不同操作。

8 原型实验与模拟结果

8.1 结果性质与原型范围

本章**不报告生产环境、主网或真实客户实绩**。第 8.3–8.6 的数值是预设工作负载下的 HIVM 合成示例，只用于验证方法和候选验收线；它们不能外推为已达成的吞吐量、安全等级、成本或 SLA。PoB、Venture Cell 与 HHA 集成尚未进入该数值实验；第 8.7 仅给出前瞻设计，不伪造结果。

设想原型闭环包含 Task Compiler、Policy Engine、双路径路由、Verification Router 与 Receipt Engine。Python 工作进程模拟模型、工具和证据核验，本地 EVM 节点模拟提交、确认与事件，不连接公链；签名采用 secp256k1，摘要采用 SHA-256，回执以规范 JSON 形成前序哈希链。参考环境为 8 核 CPU、32 GB 内存、Ubuntu 24.04、Python 3.12、Docker 27 与 Anvil；执行器限制为 2 vCPU、2 GB 内存并注入 20 ~ 120 ms 网络时延。上述版本与资源只是复现假设，不表示唯一实现要求。

8.2 工作负载、基线与统计方法

合成数据集 HIVM-Synth-10K 包含 10,000 个结构化任务：只读检索 25%、内容生成 20%、受限工具 20%、钱包/资产操作 20%、跨链或多工具编排 15%。正常样本占 70%，其余注入提示注入、参数越界、未授权工具、预算超限、证据缺失、旧 nonce 重放、回执篡改或签名替换；10% 攻击模板仅用于未见测试。数据不含真实用户材料、生产私钥或客户交易。

比较基线为：B0 Direct-Agent，仅做应用层参数检查；B1 Flat-ACL，使用静态角色—工具白名单和单次签名；B2 All-Trusted，对全部任务执行强验证与模拟锚定。它们只是控制强度基线，不映射任何外部产品。HIVM 配置加入任务级能力、参数约束、风险路由、证据核验、nonce/时窗和链式回执。性能用同机、同配额、同种子、并发 32 对比；普通模型调用计 1.00 CU，CU 是合成计算单位，不能换算成法币或实际 Gas。

每组设 30 个种子；安全实验每种子至少 1,000 次攻击，性能实验每配置运行 10,000 个任务并丢弃前 500 个预热任务。候选分析报告均值、95% bootstrap 置信区间和 p50/p95，成对比较可用 Wilcoxon 检验、Cliff’s delta 与 Holm 校正。只有公开逐种子结果后，置信区间和显著性才具有可复核意义。

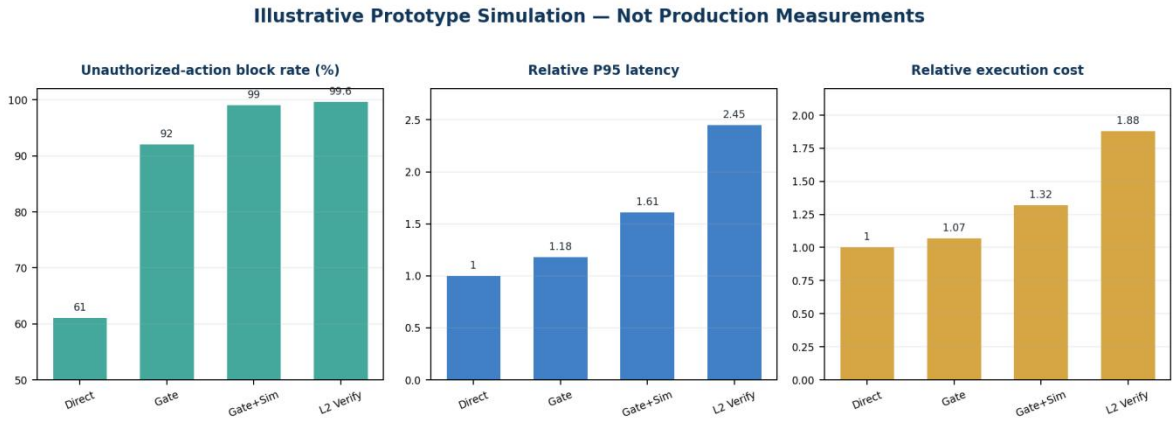

图 6 HIVM 原型模拟基准结果概览

8.3 实验一：策略门禁与越权阻断

攻击样本使用提示注入、工具别名、参数编码、跨任务能力复用和预算拆分尝试绕过门禁。攻击成功率（ASR）为“实际产生禁止副作用的攻击次数 / 全部攻击尝试次数”；越权阻断率只统计副作用前的强制拒绝，误阻断率以正常任务为分母。模型自行拒绝与系统门禁分开记录。

表 7 策略门禁模拟结果（30 种子；非生产实测）

配置	ASR（%）↓	越权阻断率（%）↑	正常任务误阻断率（%）↓
B0 Direct-Agent	41.8 ± 2.6	55.1 ± 2.8	0.7 ± 0.3
B1 Flat-ACL	13.6 ± 1.7	84.0 ± 1.9	2.1 ± 0.6
HIVM 分层门禁	2.9 ± 0.8	96.2 ± 0.9	3.4 ± 0.7

模拟中分层门禁以更高误阻断换取更低 ASR。候选验收线可设为未见模板 ASR≤5%、高风险资产动作阻断率≥95%、正常任务误阻断率≤5%；这只是未来测试的停止条件，不能据表 7 宣称系统已在真实对抗中达到该水平。任何高风险副作用都应逐例追因，不能被总体均值抵消。

8.4 实验二：双路径的时延—成本权衡

全快速配置仅执行策略检查、轻量验证和本地签名；全可信配置加入独立复核、完整证据归档和模拟锚定；自适应配置按冻结风险阈值将高资产、跨链和高不确定任务送入强路径。三者使用同一任务序列，防止样本难度差异影响比较。

表 8 双路径性能与成本模拟结果（并发 32；非生产实测）

路由策略	p50 延迟（s）↓	p95 延迟（s）↓	成本（CU/任务）↓	注入故障检出率（%）↑
全快速	0.82	1.74	1.08	62.3 ± 2.1

路由策略	p50 延迟（s）↓	p95 延迟（s）↓	成本（CU/任务）↓	注入故障检出率（%）↑
B2 全可信	2.67	5.48	2.31	96.8 ± 0.7
HIVM 自适应	1.21	3.86	1.46	91.7 ± 1.0

相对全可信，自适应示例成本降低约 37%、p50 降低约 55%，但故障检出率也降低约 5 个百分点。这正是风险路由的可见权衡，而非“同时得到最高安全和最低成本”。候选验收应额外约束高风险任务召回率≥95%、成本降幅≥25%、p95≤4.5 s，并扫描并发 1、8、32、64 观察饱和点。若低成本来自把高风险样本错分为快速路径，应判为失败。

8.5 实验三：验证等级与可信度代理

本实验只评估代理指标，不宣称测得“真实可信度”。表中 L0–L3 是方便对比的实验套餐，真实门禁仍按式（8）对每类声明检查覆盖与独立性；L4 未在本原型实现。向任务注入答案冲突、过期证据、规则违规和工具返回篡改，记录检出率、错误接受率和证据覆盖率。

表 9　分级验证代理指标模拟结果（非生产实测）

级别	故障检出率（%）↑	错误接受率（%）↓	证据覆盖率（%）↑	Brier 分数↓	增量成本（CU）
L0	38.5 ± 2.0	25.7 ± 1.8	31.2 ± 2.3	0.218	0.00
L1	70.4 ± 1.6	13.1 ± 1.3	76.8 ± 1.5	0.154	0.18
L2	90.9 ± 1.0	5.8 ± 0.9	91.5 ± 1.0	0.101	0.63
L3	96.6 ± 0.7	2.7 ± 0.6	98.1 ± 0.5	0.079	1.21

表 9 表明在合成故障上增加验证与成本同时提高检出，但“96.6% 故障检出”绝不等于“96.6% 语义正确”。正式原型还需公布各故障类型混淆矩阵、共同依赖与人工标签误差。高影响任务可把错误接受率≤5% 设为候选门槛，同时保留领域规则和人工复核。

8.6 实验四：回执篡改与重放

模拟向回执注入摘要修改、URI 替换、状态删除、旧 nonce 和跨任务复制，序列化前后篡改各占一半。普通结构化日志、单回执签名与 HIVM“签名 + 任务绑定 + 单调 nonce + 前序哈希”三种配置接受相同破坏序列；重复消费率表示同一授权产生两次效果。

表 10　回执攻击检测模拟结果（每种子 5,000 次破坏；非生产实测）

方案	篡改检出率（%）↑	重放检出率（%）↑	跨任务替换检出率（%）↑	重复消费率（%）↓	正常拒绝率（%）↓
普通结构化日志	47.6 ± 2.4	11.9 ± 1.5	18.7 ± 1.7	16.8 ± 1.6	0.2 ± 0.1

方案	篡改检出率（%）↑	重放检出率（%）↑	跨任务替换检出率（%）↑	重复消费率（%）↓	正常拒绝率（%）↓
单回执签名	98.8 ± 0.5	42.3 ± 2.2	63.1 ± 2.0	9.6 ± 1.2	0.8 ± 0.3
HIVM 链式回执	99.5 ± 0.3	98.9 ± 0.4	98.4 ± 0.5	0.6 ± 0.2	1.1 ± 0.3

模拟保留时钟漂移、nonce 竞争和崩溃恢复窗口，因而不写 100%。签名后字段篡改若未被检出，应视为阻断性缺陷。即使四项检测达到目标，回执仍只能证明签名后的完整性、来源与任务绑定，不能证明输入来自现实、密钥未泄漏或业务决定合理。

8.7 实验五：PoB 重复申领、串谋与提前结算

状态/属性测试要求 Escalate ⇒ no effect、最终验收前零结算、结算键唯一、义务金额守恒与版本不可变；对抗多 Agent 模拟注入女巫/串谋验证者、循环任务、抄袭、贿赂与声誉重置；结算混沌测试注入重复事件、未知广播、重组、托管短缺和部分跨链成功。HHA 互操作试点先验证无真实资金的 TaskPackage → BuildTask → Contribution → Receipt，再进入测试网托管。未来指标包括错误接受/拒绝率、申诉翻转率、验证者集中度、结算泄漏和对账失败；合成行为不能证明激励相容、公平性、法律合规或网络效应。

8.8 解释、稳健性与复现条件

四组结果共同支持一个有限结论：在所设合成分布中，能力门禁、风险路由、独立复核和 nonce/哈希链分别影响越权、成本、故障检出与重放，组合控制存在可观测的安全一性能权衡。稳健性测试应更换攻击模板，把网络时延放大两倍，将高风险任务占比从 10% 扫描到 50%，并绘制成本一错误接受率曲线；还应逐项消融能力令牌、独立复核和前序哈希，检验改善是否来自预期模块。

正式复现包需预先生成并哈希封存任务清单，盲化标签判定，固定容器与依赖，记录模型版本、温度、超时与重试，并输出逐种子只追加 JSONL。非确定性服务漂移必须作为实验因素。结果低于本章候选目标时，应公开差异并修订模型，不得选择性保留有利的模拟值。

9 讨论、限制与结论

9.1 讨论：从“模型可信”到“执行可问责”

HIVM 的关键转变是把研究焦点从“模型是否值得整体信任”移到“这一次动作在什么约束和证据下发生”。基础模型无法为开放世界任务提供统一正确性证明，系统却可以对主体、输入版本、可调用资源、预算、模拟条件、授权摘要、状态迁移和最终性作明确约束。这种转变不降低模型能力，而是把创造性留在认知路径，把不可逆性集中到价值路径的窄门。

“虚拟机”一词在此代表可移植的执行契约：同一任务 IR 可映射到不同模型和工具，同一策略语义可约束不同链，同一回执格式可由不同审计者验证。HIVM 不追求让所有节点复现随机推理，而追求让安全决定依赖经过规范化、冻结和验证的工件。与 EVM 的确定性执行相比，这是控制平面的虚拟化而非共识执行层的复制；它因此可以服务多链，却不能继承或替代任何目标链的安全保证。

风险自适应路由也不是简单地给任务打分后选择“快”或“慢”。式（9）优化的是可行路径，式（10）的验证、权限和预算是硬约束。高风险任务如果没有满足条件的路径，应明确失败。该设计承认可信度、成本与时延不可同时无限优化，使组织能够审计为何某类任务需要更多复核，也能发现所谓性能改善是否来自偷偷降低保证。

回执将责任分配给三个层次：发起者对目标、预算和批准负责，执行节点对所运行版本、输入输出摘要和事件负责，验证者只对声明范围内的检查负责。链上锚定提高防事后替换能力，但不把责任交给“区块链”这一抽象主体。这样的责任分解比单一“AI 可信分数”更繁琐，却能在争议时回答谁在何时基于什么做了什么。

HZB 的四段主张因而对应四类机制：“智能可调用”由 Web3 AI 产品与 HIVM 任务语义支撑；“建设可验证”由 PoB 证据、独立验收与申诉支撑；“价值可结算”由冻结义务、最终验收和冪等结算支撑；“创业可全球协同”由开放入口、Venture Cell 和可跨域验证的任务网络支撑。HIVM 是其中的受控运行时，不能独自完成后三项业务判定。

9.2 限制与开放问题

第一，形式模型依赖治理设定的风险权重、阈值和声明分类。错误配置可能使确定性策略稳定地放行危险动作；历史成功率也会受分布漂移影响。风险模型需要持续校准、反事实回放和独立审查，不能由同一个模型生成并批准自身权限。

第二，第 8 章使用合成任务和本地 EVM，不能覆盖真实拥堵、MEV、链重组、跨链桥故障、云地域故障、TEE 侧信道、密钥窃取和适应性对手。CU 不能换算为生产成本，延迟不能构成 SLA，示例置信区间也不含模型服务漂移与测量误差。只有公开复现和有限灰度数据才能逐步替换设计目标。

第三，多源复核可能存在隐藏的共同依赖。不同品牌 RPC 可能共享同一上游，不同模型可能使用相似训练数据，不同验证节点可能运行同一有缺陷的软件。未来实现需要记录证据依赖图、运营方与故障域，而不是把数量直接当作独立性。

第四，隐私与可审计性无法通过“全部上链”解决。证据最小化会降低事后复盘细度，长期保留原文又增加泄漏和合规风险。分层披露、加密保留和删除标记只是管理张力，仍需按司法辖区、数据用途和组织责任具体设计。零知识与 TEE 可减少部分披露，但不会自动解决输入真实性、密钥治理或业务合法性。

第五，HIVM 依赖外部链、钱包、桥、预言机、模型和工具。底层链最终性失败、钱包端被接管、治理串谋或允许列表误配都可能突破运行时预期。纵深防御降低单点放大效应，却不能消除系统性风险。自动化价值暴露必须有上限，并保留停止、人工接管和独立对账路径。

第六，互操作标准仍在演进。MCP、A2A 和 ERC-8004 等协议或提案可以减少集成成本，但版本、认证与安全语义可能变化；尤其 Draft 提案不能成为不可替换的信任根。HIVM 应以版本化适配器吸收变化，并在协议升级时重验能力与回执兼容性。

第七，本文没有验证 PoB 激励相容、反串谋效果、原创性/IP 判定、Venture Cell 跨法域责任、HHA 正式接口或真实资金结算。这些问题需要独立的协议试点、专业审查和法律/合规设计，不能由 HIVM 回执自动推出。

9.3 后续研究与结论

后续工作应优先完成可证伪的内核，而非扩大市场叙事：一是对 Task Compiler 的支配关系、状态机可达性、风险单调性和回执抗篡改进行属性测试与模型检查；二是公开合成数据生成器、策略、镜像和逐种子结果，接受第三方复测；三是在隔离测试网开展提示注入、RAG 投毒、沙箱逃逸、跨租户、RPC 分歧、重组、签名重放与灾难恢复演练；四是研究证据依赖图、隐私保留审计、形式化策略语言和跨链最终性组合；五是在严格限额和人工批准下采集灰度数据，并用真实失败修订风险模型。

本文提出 HIVM 这一面向 Web3 可信智能执行的分层虚拟机架构。其核心不是把大模型搬上链，也不是复制 EVM 或宣称已经部署新的 Layer 1，而是建立模型、数据、工具、钱包与多链之间的受控运行时：以结构化任务承接意图，以能力和零信任限制动作，以双路径隔离认知与价值，以分级验证匹配声明，以状态机、幂等和补偿界定效果，再以最小披露回执连接意图、证据、策略、授权和最终状态。

在这一框架中，HZB 遵循“链下形成智能，链上建立信任与价值闭环”；链上承载必要的身份、承诺、状态、结算和治理，链下承载高性能智能计算与隐私数据。HZB Network = 全球 AI 原生共建网络，Proof of Build = 核心机制，HHA = 创世应用；第一增长引擎提供可调用智能，第二增长引擎组织可验证建设。HIVM 的价值取决于未来实现能否经受复现、攻击测试、独立审查和持续治理；本文的公式、代码与合成结果是可实施、可质疑、可修订的起点，不是已部署、已通过审计或绝对安全的证明。

参考文献

[1] WOOD G. Ethereum: A Secure Decentralised Generalised Transaction Ledger: Shanghai Version[EB/OL]. (2025-02-04)[2026-08-27]. https://ethereum.github.io/yellowpaper/paper.pdf.

[2] ETHEREUM. Ethereum Virtual Machine (EVM)[EB/OL]. [2026-08-27]. https://ethereum.org/developers/docs/evm/.

[3] WILSON S. Ethereum Execution Layer Specification[EB/OL]. (2023-08-29)[2026-08-27]. https://blog.ethereum.org/2023/08/29/eel-spec.

[4] CHAINGPT. Artificial Intelligence Virtual Machine (AIVM)[EB/OL]. (2024-06-26)[2026-08-27]. https://docs.chaingpt.org/overview/learn-the-concepts/artificial-intelligence-virtual-machine-aivm.

[5] CHAINGPT. AIVM Blockchain Whitepaper[EB/OL]. (2025)[2026-08-27]. https://docs.chaingpt.org/ai-tools-and-applications/aivm-blockchain-whitepaper.

[6] TABASSI E. Artificial Intelligence Risk Management Framework (AI RMF 1.0): NIST AI 100-1[R/OL]. Gaithersburg: National Institute of Standards and Technology, 2023[2026-08-27]. https://doi.org/10.6028/NIST.AI.100-1.

[7] AUTIO C, SCHWARTZ R, DUNIETZ J, et al. Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile: NIST AI 600-1[R/OL]. Gaithersburg: National Institute of Standards and Technology, 2024[2026-08-27]. https://doi.org/10.6028/NIST.AI.600-1.

[8] LEWIS P, PEREZ E, PIKTUS A, et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks[C]//Advances in Neural Information Processing Systems 33. 2020: 9459-9474. https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html.

[9] YAO S, ZHAO J, YU D, et al. ReAct: Synergizing Reasoning and Acting in Language Models[C/OL]//The Eleventh International Conference on Learning Representations. 2023[2026-08-27]. https://openreview.net/forum?id=WE_vluYUL-X.

[10] ROSE S, BORCHERT O, MITCHELL S, et al. Zero Trust Architecture: NIST SP 800-207[R/OL]. Gaithersburg: National Institute of Standards and Technology, 2020[2026-08-27]. https://doi.org/10.6028/NIST.SP.800-207.

[11] SALTZER J H, SCHROEDER M D. The Protection of Information in Computer Systems[J]. Proceedings of the IEEE, 1975, 63(9): 1278-1308. https://doi.org/10.1109/PROC.1975.9939.

[12] DENNIS J B, VAN HORN E C. Programming Semantics for Multiprogrammed Computations[J]. Communications of the ACM, 1966, 9(3): 143-155. https://doi.org/10.1145/365230.365252.

[13] MILLER M S, YEE K P, SHAPIRO J. Capability Myths Demolished: SRL2003-02[R/OL]. Baltimore: Johns Hopkins University, 2003[2026-08-27]. https://papers.agoric.com/papers/capability-myths-demolished/abstract/.

[14] OWASP GENAI SECURITY PROJECT. OWASP GenAI LLM Top 10 2026[EB/OL]. (2026-08-03)[2026-08-27]. https://genai.owasp.org/resource/owasp-genai-llm-top-10-2026/.

[15] OWASP GENAI SECURITY PROJECT. OWASP Top 10 for Agentic Applications for 2026[EB/OL]. (2025-12-09)[2026-08-27]. https://genai.owasp.org/resource/owasp-top-10-for-agentic-applications-for-2026/.

[16] MITRE. MITRE ATLAS: Adversarial Threat Landscape for AI Systems[DB/OL]. (2021—)[2026-08-27]. https://atlas.mitre.org/.

[17] BIRKHOLZ H, THALER D, RICHARDSON M, et al. Remote ATtestation procedureS (RATS) Architecture: RFC 9334[S/OL]. IETF, 2023[2026-08-27]. https://www.rfc-editor.org/info/rfc9334/.

[18] INTEL. Attestation Services for Intel Software Guard Extensions[EB/OL]. [2026-08-27]. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html.

[19] LIU T, XIE X, ZHANG Y. zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy[C]//Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. New York: ACM, 2021. https://eprint.iacr.org/2021/673.

[20] LUNDBLADE L, MANDYAM G, O'DONOGHUE J, et al. The Entity Attestation Token (EAT): RFC 9711[S/OL]. IETF, 2025[2026-08-27]. https://www.rfc-editor.org/info/rfc9711/.

[21] HERMAN I, JONES M, SPORNY M, et al. Verifiable Credentials Data Model v2.0[S/OL]. W3C Recommendation, 2025-05-15[2026-08-27]. https://www.w3.org/TR/vc-data-model-2.0/.

[22] SPORNY M, GUY A, SABADELLO M, et al. Decentralized Identifiers (DIDs) v1.0[S/OL]. W3C Recommendation, 2022-07-19[2026-08-27]. https://www.w3.org/TR/did-core/.

[23] HERMAN I, SPORNY M, THIBODEAU T, et al. Verifiable Credential Data Integrity 1.0[S/OL]. W3C Recommendation, 2025-05-15[2026-08-27]. https://www.w3.org/TR/vc-data-integrity/.

[24] OPENTELEMETRY. OpenTelemetry Specification 1.60.0[S/OL]. [2026-08-27]. https://opentelemetry.io/docs/specs/otel/.

[25] W3C DISTRIBUTED TRACING WORKING GROUP. Trace Context[S/OL]. W3C Recommendation, 2021-11-23[2026-08-27]. https://www.w3.org/TR/trace-context/.

[26] BLOEMEN R, LOGVINOV L, EVANS J. EIP-712: Typed Structured Data Hashing and Signing[S/OL]. Ethereum Improvement Proposals, 2017[2026-08-27]. https://eips.ethereum.org/EIPS/eip-712.

[27] GIORDANO F, CONDON M, CASTONGUAY P, et al. ERC-1271: Standard Signature Validation Method for Contracts[S/OL]. Ethereum Improvement Proposals, 2018[2026-08-27]. https://eips.ethereum.org/EIPS/eip-1271.

[28] BUTERIN V. EIP-155: Simple Replay Attack Protection[S/OL]. Ethereum Improvement Proposals, 2016[2026-08-27]. https://eips.ethereum.org/EIPS/eip-155.

[29] OPENZEPPELIN. Access Control: OpenZeppelin Contracts 5.x[EB/OL]. [2026-08-27]. https://docs.openzeppelin.com/contracts/5.x/access-control.

[30] OPENZEPPELIN. Pausable: OpenZeppelin Contracts 5.x API Utilities[EB/OL]. [2026-08-27]. https://docs.openzeppelin.com/contracts/5.x/api/utils#Pausable.

[31] OPENZEPPELIN. TimelockController: OpenZeppelin Contracts 5.x Governance[EB/OL]. [2026-08-27]. https://docs.openzeppelin.com/contracts/5.x/api/governance#TimelockController.

[32] SLSA COMMUNITY. SLSA Specification v1.2[S/OL]. [2026-08-27]. https://slsa.dev/spec/v1.2/.

[33] BARKER E. Recommendation for Key Management: Part 1—General: NIST SP 800-57 Part 1 Rev. 5[R/OL]. Gaithersburg: National Institute of Standards and Technology, 2020[2026-08-27]. https://doi.org/10.6028/NIST.SP.800-57pt1r5.

[34] MODEL CONTEXT PROTOCOL PROJECT. Model Context Protocol Specification: 2026-07-28[S/OL]. (2026-07-28)[2026-08-27]. https://modelcontextprotocol.io/specification/2026-07-28.

[35] A2A PROJECT. Agent2Agent (A2A) Protocol Specification: Version 1.0.0[S/OL]. (2026)[2026-08-27]. https://a2a-protocol.org/latest/specification/.

[36] DE ROSSI M, CRAPIS D, ELLIS J, et al. ERC-8004: Trustless Agents: Draft[S/OL]. Ethereum Improvement Proposals, 2025[2026-08-27]. https://eips.ethereum.org/EIPS/eip-8004.