Tangle Network

The Operating Layer for Autonomous Work

Tangle Network Team
`hello@tangle.tools`

January 2026

Abstract

This paper introduces Tangle Network, the operating layer for autonomous work. As artificial intelligence transforms from a tool into a workforce, fundamental questions emerge about who controls the infrastructure that hosts these systems, who captures the economic value they generate, and whether access to this future will be concentrated or distributed. Tangle addresses these questions through a decentralized protocol that enables AI agents and distributed services to execute on infrastructure provided by independent, economically staked operators. The protocol introduces a blueprint system for defining arbitrary service types, a request-for-quote mechanism enabling competitive service pricing, and O(1) algorithms for staking, slashing, and reward distribution that scale to millions of participants. Developers create blueprints specifying service behavior and earn from adoption. Operators stake assets to provide compute and earn from service fees. Delegators back operators with additional stake and share in the revenue. Customers pay for services with cryptographic guarantees of accountability. This paper presents the protocol architecture, the developer platform for building expressive crypto-economic systems, and the products that demonstrate what becomes possible when AI infrastructure is owned by many rather than few.

Contents

Part I

Vision

1 Introduction

Artificial intelligence is transitioning from assistant to workforce. This transition is not hypothetical but observable in production systems deployed today. Consider what AI agents accomplished in 2025 alone: coding agents now generate 30-50% of code at major technology companies, resolving issues that previously required human engineers. Research agents synthesize literature, generate hypotheses, and design experiments across pharmaceutical and materials science domains. Trading agents execute strategies across decentralized exchanges, managing portfolios, identifying arbitrage opportunities, and rebalancing positions without human intervention. Customer service agents handle 70% of support inquiries at companies that have deployed them, escalating only the minority that require human judgment.

These systems share a common architecture: perception, reasoning, action. They observe their environment through APIs, sensors, and data feeds. They reason about objectives using large language models, retrieval-augmented generation, and planning algorithms. They take actions through tool use, API calls, and code execution. The loop runs continuously, with agents learning from outcomes and refining their approaches.

The scope of autonomous work will only expand. Agents that today require human approval for consequential actions will tomorrow operate with greater autonomy as trust accumulates. Agents that today handle narrow tasks will tomorrow handle broader ones as capabilities improve. The economic value of agent-performed work, already measured in billions of dollars annually, will grow as agents become more capable and more widely deployed.

1.1 The Infrastructure Question

The infrastructure hosting this workforce will generate enormous value, and the question of who owns that infrastructure will determine who captures that value. Three models compete.

The **centralized model** concentrates infrastructure in a small number of cloud providers. Amazon, Google, and Microsoft operate the data centers, control the APIs, and capture the margin. Developers building AI applications must trust these providers: trust that execution occurs correctly, trust that data remains private, trust that pricing remains reasonable, trust that access will not be revoked. The provider relationship is asymmetric—developers need infrastructure more than providers need any individual developer—and the terms reflect this asymmetry.

The centralized model has advantages: professional operations, high availability, economies of scale. It also has structural problems. Providers face no economic penalty for misbehavior because customers have limited alternatives. Providers capture value that developers create because the platform controls the customer relationship. Providers can change terms unilaterally because switching costs are high. These are not accidents but features of concentrated market power.

The **decentralized-compute model** distributes infrastructure across independent providers but retains centralized coordination. Projects in this category typically operate compute marketplaces where providers list resources and customers request them. Coordination occurs through a central party—a foundation, a company, a DAO—that sets terms, resolves disputes, and captures fees.

This model improves on centralization by creating competition among providers, but it does not solve the coordination problem. The central coordinator still accumulates power. Disputes still require trusted adjudication. Value still flows to the platform rather than to participants.

The **cryptoeconomic model** replaces trusted coordination with economic mechanisms. Providers stake assets that can be slashed for misbehavior, creating accountability without trust. Smart contracts encode rules that execute automatically, eliminating human discretion in dispute resolution. Token governance distributes decision-making to stakeholders, preventing power accumulation. This is the model Tangle implements.

1.2 Concrete Scenarios

To make the infrastructure question concrete, consider three scenarios that illustrate where current models fail and where cryptoeconomic infrastructure succeeds.

Scenario 1: The Trading Agent. An institutional investor deploys an AI agent to execute trading strategies across decentralized exchanges. The agent manages $100 million in assets, monitoring market conditions, identifying opportunities, and executing trades. The agent runs on cloud infrastructure provided by a major tech company.

In the centralized model, the cloud provider can observe the agent's trading patterns, positions, and strategies. Nothing prevents an employee, a compromised system, or the provider itself from front-running the agent's trades. The investor has no cryptographic guarantee that execution occurred correctly, no recourse if the provider extracts value through information asymmetry, and no audit trail proving what actually happened.

In the Tangle model, the agent runs on operator infrastructure where execution produces cryptographic attestations. Operators stake assets that can be slashed for information leakage or front-running. Multiple operators can verify execution through redundant computation. The investor has on-chain evidence of what occurred, economic recourse for misbehavior, and the ability to choose operators based on reputation and stake.

Scenario 2: The Research Agent. A pharmaceutical company deploys AI agents to analyze clinical trial data, identify promising compounds, and design follow-up experiments. The agents process proprietary data worth billions in competitive advantage. The agents run on a decentralized compute network.

In the decentralized-compute model, the network operator coordinates job dispatch and result collection. Providers see the data they process. The network operator can observe which data flows where. Disputes about incorrect results require human adjudication by a party that may have conflicts of interest.

In the Tangle model, blueprints define verification mechanisms: perhaps trusted execution environments for data isolation, perhaps secure multi-party computation for analysis without revealing inputs, perhaps redundant execution with cryptographic comparison of outputs. Slashing conditions specify penalties for data leakage. The pharmaceutical company selects operators based on security practices and stake. Disputes resolve through on-chain mechanisms rather than human judgment.

Scenario 3: The Coding Agent. A software company deploys AI agents to maintain legacy codebases, resolve bugs, and implement features. The agents have access to source code representing years of development effort. The agents run on the company's cloud provider.

In the centralized model, the cloud provider has contractual obligations to protect customer data, but no economic penalty for breach. If the provider's security fails and source code leaks, the company's recourse is litigation—slow, expensive, and uncertain. The provider's incentive is to minimize security spending up to the point where expected litigation costs exceed the savings.

In the Tangle model, operators stake assets proportional to the value of code they access. A breach triggers slashing that destroys the operator's stake. The economic penalty for security failure is immediate, automatic, and proportional. Operators have direct financial incentive to invest in security up to the value of their stake. The company selects operators whose stake meets or exceeds the value of the code being processed.

1.3 Why Now

Several converging trends make decentralized AI infrastructure both possible and necessary at this moment.

Agent capabilities have reached commercial relevance. AI agents in 2020 were research curiosities. AI agents in 2025 generate measurable business value. This transition creates demand for infrastructure that matches the scale and stakes of commercial deployment.

Cryptoeconomic mechanisms have proven at scale. Proof-of-stake networks secure hundreds of billions of dollars. Restaking protocols manage tens of billions in additional security. The mechanisms—staking, slashing, governance—are no longer experimental but proven. Applying them to AI infrastructure is engineering rather than research.

Regulatory pressure on centralized providers is increasing. Antitrust scrutiny, data sovereignty requirements, and AI-specific regulations create friction for centralized infrastructure. Decentralized alternatives face lower regulatory risk because they distribute rather than concentrate power.

Developer demand for ownership is growing. The web2 pattern—developers create value, platforms capture it—has created a generation of developers seeking alternatives. Protocols that distribute value to creators attract talent that centralized platforms cannot.

The window for building foundational infrastructure is finite. Once patterns establish, they become difficult to change. The time to build decentralized AI infrastructure is before centralized infrastructure becomes entrenched, not after.

1.4 From Scenarios to Requirements

The scenarios above reveal concrete requirements that infrastructure must satisfy.

The **trading agent scenario** requires cryptographic attestation that execution occurred correctly, economic recourse if operators leak information, and the ability to choose operators based on security practices. These requirements demand *programmable verification* (blueprints can define how to verify execution) and *meaningful slashing* (operators have stake at risk proportional to the value they might extract).

The **research agent scenario** requires data isolation, confidential computation, and dispute resolution that does not depend on conflicted parties. These requirements demand *expressive service definitions* (blueprints can specify TEE requirements, MPC protocols, or other isolation mechanisms) and *on-chain dispute resolution* (slashing proposals with evidence and dispute windows).

The **coding agent scenario** requires security investment proportional to code value, immediate economic consequences for breach, and transparent audit trails. These requirements demand *configurable exposure* (operators stake proportional to the value they access) and *scalable accounting* (the system handles millions of positions without prohibitive gas costs).

These requirements (programmable verification, expressive service definitions, meaningful slashing, configurable exposure, and scalable accounting) drove Tangle's design.

1.5 Tangle's Approach

Tangle Network is the operating layer for autonomous work: the protocol that enables AI agents and distributed services to execute on infrastructure provided by independent, economically staked operators.

The protocol introduces three core innovations that address these requirements. **Blueprints** are reusable templates that define service types. A blueprint specifies what computation the service performs, how it should be priced, what verification mechanisms apply, and what slashing conditions govern operator behavior. Developers create blueprints; the protocol handles deployment and economics. **O(1) algorithms** enable the protocol to scale to millions of participants. Share-based accounting handles delegation without iterating over positions. Accumulated-per-share rewards handle distribution without iterating over recipients. Proportional slashing adjusts exchange rates without iterating over stakes. **Expressive hooks** enable blueprints to customize behavior at every lifecycle stage. Custom validation for operator registration, custom logic for service activation, custom verification for job completion—blueprints implement the specifics while the protocol handles the commons.

Every participant earns from the value they create. Developers earn from blueprint adoption through both fee splits and inflation rewards. Operators earn from service fees based on the compute they provide. Delegators earn from backing reliable operators, sharing in operator rewards proportional to their stake. Customers pay for services and receive cryptographic guarantees of accountability. The protocol captures a governance-controlled fee (currently 10%) that funds ongoing development rather than enriching a central party.

1.6 Paper Organization

The remainder of this paper proceeds as follows. Section 2 articulates the vision driving Tangle's design: the three-layer architecture, the design philosophy, and the future of AI infrastructure. Part II presents the protocol mechanics: blueprints and services, operators and staking, the service marketplace, fee distribution, inflation, slashing, and governance. Part III describes the developer platform: the hook system enabling expressive crypto-economic customization, the Blueprint SDK for building services, and approaches to verifiability and detection. Part IV presents the products built on Tangle: the decentralized sandbox runtime and the agentic workbench. Part V concludes with roadmap and summary.

2 The Operating Layer for Autonomous Work

Tangle's vision is to provide the shared operating layer where autonomous work is authored, executed safely by a network of staked operators, and settled through on-chain coordination. This section articulates the architecture, design philosophy, and the future this infrastructure enables.

2.1 Three-Layer Architecture

Tangle unifies three traditionally separate concerns into one coherent platform.

The **Agentic Workbench** provides the environment where workflows are designed through collaboration between humans and AI agents. Teams iterate on autonomous processes with clear success criteria before deployment. The workbench handles the creative and design aspects of autonomous work: defining what agents should do, testing their behavior, and refining their performance based on evidence.

The **Decentralized Sandbox Runtime** provides the execution environment where workloads run safely on operator infrastructure. Independent operators host containers with process, filesystem, and network isolation. Every execution produces structured logs enabling verification and improvement. The runtime handles the operational aspects: actually running the agents, enforcing resource limits, and maintaining the security boundaries that make autonomous execution safe.

The **On-Chain Protocol** provides the coordination layer where operators register capabilities, customers request services, payments flow to participants, and misbehavior triggers economic penalties. Smart contracts encode the rules; cryptography enforces them. The protocol handles the economic aspects: who gets paid, how much, and under what conditions.

These layers are designed to function independently while composing into a complete system. Developers can use the protocol without the workbench. Operators can run infrastructure without building their own interfaces. Customers can access services without understanding the underlying mechanics. Yet all three layers share common primitives: blueprints define services, operators execute them, and the protocol settles the economics.

2.2 Design Philosophy

Protocol design involves countless decisions, each with tradeoffs and alternatives. This section articulates the six principles that guide Tangle's choices, explaining not just what each principle means but why it matters, how it manifests in concrete design decisions, and what tensions arise when principles conflict.

2.2.1 Democratized Ownership

The dominant model for AI infrastructure concentrates value extraction in platform operators. Cloud providers capture margin on every API call. Data centers extract rent from physical resources. Platform operators retain the relationship with end customers, reducing developers and operators to commoditized inputs. This extraction model creates misaligned incentives: platforms profit from lock-in, opacity, and switching costs rather than from creating genuine value.

Tangle inverts this model. Value flows to those who create it: developers earn from blueprints they create, operators earn from compute they provide, delegators earn from stake they commit. The protocol itself captures only a governance-controlled fee (currently 10%) that funds ongoing development. This fee is not extracted rent but payment for the public good of protocol coordination.

This principle manifests throughout the protocol. The inflation distribution explicitly allocates 25% to developers and 25% to operators, ensuring that both blueprint creators and service providers share in network growth. The payment split system allows blueprints to define custom distributions, but always routes value to the participants who earned it. The governance system grants voting power proportional to stake, ensuring that those with economic skin in the game control protocol evolution.

The alternative approach—platform-captured value—has precedent in web2 infrastructure where cloud providers captured most of the value from the mobile and internet booms. Tangle rejects this model not on ideological grounds but practical ones: participants who capture value from their contributions have stronger incentives to contribute quality work, attract more participants, and build sustainable ecosystems.

2.2.2 Permissionless Participation

Permission systems create gatekeepers. Gatekeepers accumulate power. Power corrupts. This is not cynicism but observation: every permissioned system eventually serves the interests of those who control permissions rather than those who need access.

Tangle implements permissionless participation at every layer. Anyone can become an operator by staking the minimum bond and running the software. Anyone can become a developer by deploying a blueprint. Anyone can become a customer by paying for services. Anyone can become a delegator by backing operators. No committee approves applications. No foundation decides who may participate. No terms of service exclude competitors.

Abuse prevention uses economic rather than administrative mechanisms. The minimum stake bond creates Sybil resistance without identity verification. Schema validation ensures well-formed services without human review. Slashing punishes misbehavior without requiring prior permission.

Permissionless systems cannot prevent low-quality operators from registering. Tangle addresses this through market mechanisms: customers choose operators based on reputation, stake, and history. Bad operators lose customers; good operators attract delegations. The market filters quality more effectively than any committee could.

2.2.3 Developer Expressiveness

Protocols face a fundamental choice: provide specific functionality or provide general primitives. Specific functionality is easier to use but limits what developers can build. General primitives are harder to use but enable arbitrary applications.

Tangle chooses expressiveness. The protocol provides primitives—blueprints, services, jobs, payments, slashing—that developers compose into specific applications. The hook system enables custom logic at every lifecycle stage: before operator registration, after service activation, during job execution, upon termination. Blueprints define their own pricing models, membership rules, slashing conditions, and verification mechanisms.

This expressiveness enables use cases the protocol designers did not anticipate. A blueprint for AI inference might implement optimistic execution with fraud proofs. A blueprint for data processing might implement redundant execution with majority voting. A blueprint for cryptographic services might implement threshold signatures with MPC. The protocol does not need to understand these verification mechanisms; it only needs to execute the hooks that blueprints define.

The cost of expressiveness is complexity. Developers must understand the hook system and handle edge cases the protocol cannot anticipate. This tradeoff suits a protocol targeting sophisticated developers building production infrastructure.

The hook system implements the principle directly. The protocol calls developer-defined functions at each lifecycle stage: `onRegister` for operator validation, `onRequest` for service validation, `onApprove` for approval processing, `onSlash` for custom slashing logic. Each hook is optional; blueprints implement only what they need.

2.2.4 Economic Security

Trust is expensive. Legal agreements require enforcement mechanisms. Reputation requires long histories. Centralized authorities require audits. These mechanisms work, but they impose costs that scale poorly and exclude participants who lack resources for compliance.

Cryptoeconomic security provides an alternative: trust through stake. Operators deposit assets that can be destroyed if they misbehave. The cost of cheating exceeds any benefit from cheating. Trust emerges not from reputation or authority but from aligned incentives.

Tangle implements economic security through several mechanisms. Operators stake assets proportional to their participation. Services specify exposure requirements—what fraction of operator stake backs that service. Slashing proposals require evidence and survive a dispute window. Executed slashes reduce operator stake and, through share-based accounting, delegator stake proportionally.

The security condition is explicit: for any attack, the expected cost (probability of detection multiplied by slash amount) must exceed the expected benefit (value gained from successful attack). Blueprints configure these parameters based on their threat models. High-value services require higher exposures. Services with reliable detection mechanisms can accept lower exposures. The protocol provides the framework; blueprints configure the specifics.

The 7-day dispute window allows operators to contest slashing proposals. Governance can cancel unjust slashes. The pending slash mechanism blocks withdrawals during the dispute period, preserving security while maintaining fairness.

Economic security is not absolute security. An attacker willing to lose their stake can still attack. The design assumption is that rational actors will not attack when the expected cost exceeds the expected benefit. Against irrational attackers or state-level adversaries with unlimited resources, economic security provides weaker guarantees. Tangle is designed for the rational adversary model that applies to most commercial applications.

2.2.5 Scalable Efficiency

Gas costs determine what is practically possible on EVM chains. An operation costing 100,000 gas is routine. An operation costing 10,000,000 gas is expensive. An operation costing 1,000,000,000 gas is impossible. Protocol design must respect these constraints.

Naive implementations of staking, slashing, and reward distribution iterate over participants. Distributing rewards to N delegators requires N storage writes, costing approximately $N \times 5000$ gas. For 1 million delegators, this exceeds any block gas limit. The protocol would be theoretically sound but practically unusable.

Tangle implements O(1) algorithms that handle millions of participants without iteration. Share-based accounting tracks delegation through exchange rates rather than individual balances. Accumulated-per-share rewards track distributions through a single accumulator rather than per-user updates. Proportional slashing adjusts exchange rates rather than iterating over positions.

The share-based accounting system illustrates the approach. When a delegator stakes amount a, they receive shares $s = a \times \frac{T_s+V}{T_a+V}$, where T_s is total shares, T_a is total assets, and V is a virtual offset preventing first-depositor attacks. The exchange rate $r = T_a/T_s$ determines the value of each share. When slashing reduces total assets, the exchange rate decreases, and all delegators' positions lose value proportionally—without any iteration over individual positions.

This efficiency enables Tangle to scale. A protocol that cannot handle millions of delegators cannot support the volume of AI infrastructure that the autonomous future will require. O(1) algorithms are not premature optimization; they are fundamental design requirements.

The tradeoff is implementation complexity. Share-based accounting requires careful handling of dust (rounding errors), virtual offsets (preventing manipulation), and precision (avoiding overflow). These complexities are contained within the protocol implementation; developers and users interact with simple interfaces that hide the underlying mechanics.

2.2.6 Isolation by Default

AI agents present unique security challenges. Traditional software executes predictable code paths that developers explicitly wrote. AI agents execute actions that emerge from model weights and training data—actions that may surprise even their creators. An agent told to "optimize revenue" might discover that revenue increases when competitor systems are unavailable. An agent managing infrastructure might learn that provisioning resources prevents failures, leading to unbounded resource consumption.

Isolation by default addresses this uncertainty. Sandboxes enforce explicit permissions: agents cannot access network resources, filesystems, or system calls without explicit grants. Containers separate processes, preventing escape from the sandbox. Resource limits prevent exhaustion attacks. Structured logging captures all agent actions for audit and improvement.

This principle distinguishes Tangle from infrastructure designed for traditional workloads. Cloud compute assumes that developers understand what their code will do. Agent infrastructure assumes that developers have limited knowledge of what their agents will do. The security model must accommodate this uncertainty.

The sandbox implementation enforces isolation through multiple layers. Process isolation prevents cross-container access. Filesystem isolation provides each container its own root. Network isolation controls ingress and egress. Capability-based permissions grant specific resources (read this file, access this API, use this amount of CPU) rather than broad privileges. An agent operating within these boundaries can cause limited harm even if its behavior surprises its creator.

The cost of isolation is functionality. Agents that cannot access the network cannot fetch external data. Agents that cannot write to the filesystem cannot persist state. Blueprints must explicitly grant the capabilities their agents need, creating a tradeoff between security and functionality. The default is secure; developers opt into risk by granting capabilities.

2.2.7 Tensions Between Principles

Principles sometimes conflict. Permissionless participation allows low-quality operators; democratized ownership cannot exclude them. Scalable efficiency requires algorithmic complexity; developer expressiveness wants simplicity. Economic security requires locked stake; permissionless participation wants minimal barriers.

Tangle navigates these tensions through careful mechanism design. Permissionless participation combined with market selection allows anyone to enter while ensuring quality emerges. Scalable efficiency implemented behind simple interfaces provides both scale and usability. Economic security with flexible exposure requirements lets blueprints configure the tradeoff between accessibility and safety.

The most significant tension is between expressiveness and security. Hooks that enable arbitrary custom logic also enable bugs, vulnerabilities, and unexpected interactions. Tangle addresses this by providing safe defaults, validating hook inputs, and isolating hook failures from protocol state. Hooks cannot corrupt protocol accounting; they can only affect their own blueprint's behavior. This bounded blast radius lets developers experiment while protecting the broader ecosystem.

Design philosophy is not a checklist but a compass. When facing novel decisions, these principles guide toward consistent choices. When principles conflict, understanding their rationale enables informed tradeoffs. The result is a protocol that embodies coherent values rather than ad-hoc accumulation.

2.3 The Future of AI Infrastructure

The trends driving Tangle's design will only accelerate. AI agents will become more capable, handling increasingly complex tasks with less human oversight. The volume of agent-driven computation will grow, creating demand for infrastructure that scales. The economic value generated by agents will increase, making the question of who captures that value more consequential.

In this future, the infrastructure layer becomes foundational. Just as cloud providers became essential to the internet economy, AI infrastructure providers will become essential to the autonomous economy. The difference is whether this infrastructure will be owned by a few corporations or distributed across a network of independent operators.

Tangle enables the distributed path. By creating economic incentives for operators to provide reliable compute, cryptographic accountability for misbehavior, and governance mechanisms for collective decision-making, the protocol establishes infrastructure that can grow without centralizing. More operators mean more capacity, not more concentration. More usage means more value distributed, not more value extracted.

This is not merely an ideological position; it is a practical one. Decentralized infrastructure is more resilient to single points of failure. Competitive markets produce better pricing than monopolies. Cryptographic verification provides stronger guarantees than legal agreements. The same properties that made decentralized finance compelling make decentralized AI infrastructure compelling.

The operating layer for autonomous work must be built now, while the patterns are still forming. Tangle is that layer.

Part II

Protocol

Part I presented the vision: what Tangle aims to achieve and why it matters. This part describes how the protocol actually works. We begin with blueprints and services, the fundamental abstractions through which developers define and customers consume computational work. We then examine operators and staking, the economic foundation that provides security and accountability. The service marketplace describes how customers find operators and negotiate pricing. Fee distribution and inflation explain how value flows to participants. Slashing provides the enforcement mechanism that makes economic security meaningful. Finally, governance describes how the protocol evolves under stakeholder control.

Each section builds on previous ones. Understanding blueprints helps understand why operators stake. Understanding staking helps understand why slashing matters. By the end of this part, readers should understand not just what each mechanism does but how they compose into a coherent system.

3 Blueprints and Services

The blueprint system forms the foundation of Tangle's service architecture. Blueprints define what services do; services are running instances of blueprints with assigned operators. This section describes both abstractions and their lifecycle.

3.1 What Is a Blueprint?

A blueprint is a reusable template defining a service type. Developers create blueprints to specify the computational tasks a service performs, how the service should be priced, what requirements operators must meet, and optionally, custom logic governing service behavior.

Formally, a blueprint comprises metadata for discovery (name, description, author), configuration for pricing and membership, job definitions specifying the tasks operators execute, validation schemas ensuring well-formed operator registrations and service requests, and an optional custom service manager contract implementing hooks for advanced behavior.

Blueprints are identified by unique numeric IDs assigned at creation. Once an operator registers for a blueprint, the protocol locks the blueprint's metadata, preventing developers from modifying terms after operators have committed. This lock protects against bait-and-switch attacks where a developer might change a blueprint's behavior after attracting operators. The blueprint remains active until the owner deactivates it, though existing services continue operating regardless of blueprint status.

3.2 What Is a Service Instance?

A service is a running instance of a blueprint with assigned operators. Where blueprints define what a service type does, service instances represent actual deployments serving actual customers.

A service comprises a reference to its blueprint, a set of participating operators with their committed exposures, configuration parameters validated against the blueprint's schema, a time-to-live (TTL) defining service duration, and payment configuration specifying token and amount.

Services move through defined states. A pending service awaits operator approvals. An active service is operational, accepting jobs and billing customers. A terminated service has completed its lifecycle, with remaining escrow refunded and operators released.

3.3 Service Lifecycle

The lifecycle proceeds through request, approval, activation, operation, and termination.

A customer initiates service creation by submitting a request specifying the blueprint, desired operators, configuration, payment, time-to-live, permitted callers authorized to submit jobs, and security requirements defining minimum stake exposures. The protocol validates parameters, creates a pending request, and notifies the requested operators.

Each operator must respond to the pending request. Approval requires committing an exposure in basis points, indicating what fraction of the operator's stake backs this service. If any operator rejects the request, the service fails and payment refunds to the customer. When all operators approve, the service activates automatically.

Upon activation, operators begin providing the service. Customers submit jobs, operators execute them and submit results, and the payment system distributes funds according to the configured model. For subscription services, the protocol bills at each interval. For event-driven services, payment accompanies each job submission.

Services terminate through customer request, time-to-live expiration, or failure to maintain minimum operator count. Upon termination, remaining escrow refunds to the customer, operators may unstake their committed exposure, and the service manager receives a termination callback.

3.4 Pricing Models

Blueprints support three pricing models to accommodate different service types.

The **pay-once** model collects a single upfront payment at request time, distributed to participants when all operators approve. This suits one-time computations or short-duration services where the cost is known in advance.

The **subscription** model collects recurring payments at fixed intervals from a customer-funded escrow account. Customers deposit funds; the protocol withdraws at each billing cycle. This suits ongoing services with predictable costs.

The **event-driven** model collects payment per job execution. Customers pay when they submit jobs, with the amount specified in the job definition. This suits variable workloads where usage determines cost.

3.5 Membership Models

Services can use either fixed or dynamic membership.

Fixed membership locks the operator set at service creation. Operators cannot join or leave once the service activates. This model provides stability for services requiring consistent, predetermined participants.

Dynamic membership permits operators to join after activation and leave subject to exit queue constraints. Operators schedule exits, wait through a configurable queue period, and then execute the exit. This model enables long-running services to adapt to changing conditions, adding operators as demand grows or replacing operators who become unreliable.

4 Operators and Staking

With blueprints and services established as the protocol's fundamental abstractions, we turn to the entities that make services operational: operators. Operators are staked entities that run services and earn rewards. The staking system provides the economic security backing all service guarantees. This section describes operator registration, delegation, exposure selection, and the share-based accounting that enables scalable operations.

4.1 Operator Registration

An entity becomes an operator by depositing a bond meeting the minimum stake requirement for their chosen asset. Registration creates operator metadata tracking self-stake amount, delegation count, operational status, and leaving schedule if applicable. Operators begin in Active status, eligible to participate in services.

Active operators then register for specific blueprints they wish to serve. Blueprint registration requires providing an ECDSA public key for gossip network identity, an RPC endpoint for off-chain communication, and optional registration inputs validated against the blueprint's schema. The protocol validates that the blueprint is active, the operator status is Active, minimum stake requirements are met, the key format is correct (65-byte uncompressed ECDSA), and schema validation passes.

Operators may register for multiple blueprints, enabling diversification across service types. An optional limit constrains the maximum blueprints per operator, preventing resource spreading that might compromise service quality.

4.2 Delegation Modes

Operators configure their delegation mode, determining who may provide additional stake.

Disabled mode accepts only self-stake. No external delegations are permitted. This suits operators who prefer full control over their stake composition.

Whitelist mode accepts delegations only from addresses the operator has explicitly approved. This suits operators with known partners or institutional relationships.

Open mode accepts delegations from any address. This suits operators seeking to maximize their stake base and the rewards that flow from larger stake.

Changing delegation mode does not affect existing delegations. Mode changes only govern new delegation attempts.

4.3 Exposure Selection

When operators approve service requests, they commit an exposure in basis points (0 to 10000, representing 0% to 100%). This exposure determines what fraction of the operator's total stake backs the specific service.

Exposure has two consequences. First, it determines reward share: operators with higher exposure receive proportionally more of the service's operator payment allocation. Second, it bounds slashing: operators can lose at most their committed exposure percentage, regardless of slash severity.

This design enables operators to participate in multiple services with bounded total risk. An operator with 1000 TNT stake might commit 20% exposure to Service A, 30% to Service B, and 10% to Service C. Even if all three services experienced maximum slashing events, the operator's loss would be bounded by these commitments.

Service requesters can specify minimum and maximum exposure requirements per asset. Operators must commit within these bounds when approving. This ensures services receive adequate security backing while preventing operators from over-committing.

4.4 Blueprint Selection for Delegators

Delegators choose how their stake participates across the operator's blueprints.

All mode means the delegation backs all blueprints the operator currently serves and any blueprints they register for in the future. This maximizes exposure and potential rewards but means delegators share in slashing from any of the operator's services.

Fixed mode means the delegation backs only specific blueprints the delegator selects. This limits exposure to known service types but means delegators may miss rewards from blueprints they did not select.

Delegators can add or remove blueprints from their fixed selection. They cannot remove the last blueprint (which would make the delegation undefined); they must withdraw instead.

4.5 O(1) Share-Based Accounting

The staking system uses share-based accounting, enabling O(1) operations for deposits, withdrawals, and value changes across unlimited participants.

Each operator maintains a reward pool tracking total shares issued and total underlying assets. When delegating amount x to an operator with pool state (T_s, T_a), the delegator receives shares:

$$h = x \cdot \frac{T_s + V}{T_a + V}$$

where V is a virtual offset preventing first-depositor manipulation attacks.

The delegation's current value is $v = h \cdot (T_a/T_s)$. When pool assets change through rewards or slashing, this value changes proportionally without per-delegator storage updates. This is the key insight enabling scalable operations.

When slashing occurs, only total assets decrease. Shares remain constant. The exchange rate automatically decreases, affecting all delegators proportionally. No iteration is required regardless of delegator count. This enables Tangle to slash millions of delegators in a single storage write.

5 The Service Marketplace

Operators now exist and are staked, but how do customers find them and agree on prices? The service marketplace bridges supply (operators) with demand (customers), enabling competitive pricing and efficient service creation. This section describes the request-for-quote system, its design rationale, market dynamics, security properties, and comparison to alternative mechanisms.

5.1 Why RFQ Over Alternatives

Service pricing mechanisms face a fundamental design choice: how should prices be determined? Several alternatives exist, each with tradeoffs that informed Tangle's choice of request-for-quote.

Fixed pricing sets prices in the blueprint or by governance. This is simple but inflexible. Fixed prices cannot adapt to changing costs, demand fluctuations, or operator differentiation. When compute costs drop, customers overpay. When demand spikes, operators are undersupplied. Fixed pricing creates inefficiency.

Continuous order books match bids and asks in real-time. This works well for fungible assets but poorly for services. Services are heterogeneous: operators differ in reliability, geographic location, hardware specifications, and reputation. An order book that treats operators as interchangeable misses important quality dimensions.

Automated market makers (AMMs) use algorithmic pricing based on liquidity pools. AMMs excel for fungible token swaps but struggle with services. What would a service AMM's liquidity pool contain? How would it price operator-specific quality differences? The abstraction does not fit.

Auctions collect bids and select winners algorithmically. Auctions can work for services but introduce latency (auction periods) and complexity (auction mechanism design). For services where customers want specific operators, auctions are unnecessarily indirect.

Request-for-quote asks operators to provide binding prices for specific requests. The customer collects quotes, evaluates them holistically (price, reputation, other factors), and selects. RFQ accommodates operator heterogeneity naturally. It requires no on-chain order book state. It provides instant finality once quotes are collected. It allows operators to customize pricing based on the specific request (service duration, resource requirements, current capacity).

Tangle chose RFQ because it best fits the service marketplace's requirements: heterogeneous providers, customer-specific requirements, and the need for both flexibility and speed.

5.2 RFQ Tradeoffs and Limitations

Intellectual honesty requires acknowledging RFQ's drawbacks alongside its advantages.

Off-chain infrastructure dependency. RFQ requires operators to run quote-serving software continuously. If an operator's quote server is down, they receive no service requests regardless of their on-chain registration. This creates a liveness dependency absent in purely on-chain mechanisms. A fixed-price blueprint continues operating even when individual operators are offline; an RFQ blueprint requires active operator participation at the quote stage.

Information asymmetry. Operators see request parameters before committing to prices. A sophisticated operator could analyze request patterns, customer addresses, or service configurations to infer urgency and adjust pricing accordingly. This is price discrimination in the economic sense: different customers may receive different prices for equivalent service. The protocol does not prevent this; it considers it market dynamics rather than manipulation.

Quote spam and griefing. Operators may sign quotes with no intent to fulfill, wasting customer effort when they attempt submission. Conversely, customers may flood operators with quote requests while accepting none, consuming operator resources. The protocol mitigates these through reputation (unreliable parties are avoided) and soft rate limits, but determined griefers can impose costs on counterparties.

Aggregation inefficiency. Unlike AMMs that aggregate liquidity into continuous price curves, RFQ provides discrete quotes at specific moments. A customer seeking service at 3 AM when most operators are offline may receive worse prices than one seeking service at peak hours. RFQ does not smooth liquidity across time; it reflects instantaneous market conditions.

Quote shadowing. Operators monitoring P2P traffic can observe competitors' quotes and adjust their own prices accordingly. An operator who consistently undercuts competitors by small margins extracts value without contributing price discovery. This is arbitrage-like behavior that RFQ cannot prevent, though its efficiency effects are ambiguous (it may tighten spreads while reducing operator margins).

These tradeoffs are acceptable for Tangle's use case. The benefits of heterogeneous pricing, instant settlement, and minimal on-chain state outweigh the costs of infrastructure requirements, information asymmetry, and aggregation limitations. Blueprints with different requirements (fungible services, latency sensitivity, liquidity depth) might warrant different mechanisms; the hook system allows such customization.

5.3 The Request for Quote System

The RFQ flow separates negotiation from settlement. Negotiation happens off-chain, where communication is fast and free. Settlement happens on-chain, where commitments become binding.

Operators register for blueprints with ECDSA public keys and RPC endpoints. They run off-chain software that listens for quote requests on the peer-to-peer network or via direct RPC calls. When a quote request arrives, the operator evaluates resource requirements, checks availability, determines pricing, and returns a signed quote.

A quote contains the blueprint ID, service duration in blocks (TTL), total cost in the payment token, timestamp when the quote was generated, expiry timestamp after which the quote becomes invalid, and security commitments specifying the operator's exposure per required asset. The operator signs this structured data using EIP-712 typed data signatures, binding the quote to the specific chain and contract.

Customers collect quotes from their desired operators, verify the terms are acceptable, and submit all quotes in a single transaction. The protocol verifies each signature, checks that quotes are not expired, validates that quotes have not been previously used (replay protection), confirms all operators are registered for the blueprint, and ensures the customer has provided sufficient payment. If all checks pass, the service activates immediately.

5.4 Off-Chain Coordination

The quote collection process occurs through Tangle's peer-to-peer network, built on libp2p. Understanding this off-chain layer is essential to understanding how the marketplace functions in practice.

Quote request propagation uses gossip protocols. A customer publishes a quote request specifying the blueprint, desired operators, service parameters, and deadline. The request propagates through the network, reaching registered operators who evaluate and respond.

Direct RPC queries provide an alternative for customers who know which operators they want. Each operator's RPC endpoint is stored on-chain at registration. Customers can query operators directly, bypassing gossip entirely. This path is faster but requires knowing operator endpoints.

Spam prevention relies on rate limiting and reputation. The P2P network limits message rates per peer. Operators may deprioritize or ignore requests from addresses with poor reputation (failed services, disputed payments). These soft limits prevent network flooding without requiring on-chain enforcement.

Quote aggregator services may emerge as ecosystem infrastructure. An aggregator collects quotes from multiple operators on behalf of customers, presenting curated options. Aggregators compete on quote quality, response time, and operator coverage. The protocol does not privilege any aggregator; they are ordinary network participants.

5.5 Quote Verification and Security

The protocol implements multiple verification layers to ensure quotes are binding, current, and resistant to manipulation.

EIP-712 signatures bind quotes to specific chains and contracts, preventing cross-chain replay. The domain separator includes the chain ID and verifying contract address. A quote signed for Tangle on Ethereum cannot be replayed on Tangle on Arbitrum.

Each quote includes a unique timestamp; the protocol tracks used quotes by their digest, preventing same-chain replay. Once a quote is used to create a service, its digest is marked; attempting to reuse it fails with `QuoteAlreadyUsed`.

Quotes expire after their specified expiry timestamp, limiting the window for use. An operator who signs a quote at time t with expiry $t + 1$ hour knows the quote cannot bind them beyond that hour. This bounds operator commitment duration.

The protocol enforces a maximum quote age (default one hour), ensuring quotes reflect current operator availability and pricing. Even if an operator sets a long expiry, the protocol rejects quotes whose timestamp predates the maximum age. This prevents customers from stockpiling quotes when prices are low.

5.6 MEV and Front-Running Considerations

Blockchain transactions are visible in mempools before inclusion, creating opportunities for MEV (maximal extractable value) extraction. The RFQ system's security under MEV pressure merits analysis.

Quote sniping would involve an attacker observing a customer's pending transaction, extracting the quotes, and front-running with their own transaction using those quotes. This attack fails because quotes specify security commitments (operator exposures) that the attacker cannot provide. Quotes are bound to specific operators; only those operators' registrations satisfy the validation.

Price manipulation would involve an attacker colluding with operators to offer artificially high quotes, capturing the price difference. This attack is mitigated by competition: customers can always seek quotes from other operators. An operator cartel would need to control all operators for a blueprint to manipulate prices, at which point the attack is indistinguishable from legitimate market power.

Sandwich attacks (front-running and back-running a transaction to extract value) have limited applicability. RFQ transactions do not interact with liquidity pools or price oracles in ways that create sandwich opportunities. The transaction creates a service at fixed, pre-agreed prices; there is no price curve to manipulate.

The primary MEV risk is **transaction censorship**: a block producer might delay or exclude RFQ transactions to benefit a competing customer or operator. This is a general blockchain challenge, not specific to RFQ. Mitigation includes using private mempools, MEV-protected relays, or chains with fair ordering guarantees.

5.7 Failure Modes and Recovery

Real-world systems experience failures. The RFQ system's behavior under failure conditions determines its robustness.

Operator unavailable after quoting: An operator signs a quote but becomes unavailable before the customer submits the transaction. The customer's transaction succeeds (creating the service), but the operator cannot fulfill their obligations. This is handled through slashing: the absent operator fails heartbeats, violates SLA conditions, and faces economic penalties. Customers may also terminate services with non-performing operators and receive refunds.

Quote expiry during transaction: A customer collects quotes, but by the time their transaction is included, one or more quotes have expired. The transaction fails with `QuoteExpired`. The customer must collect fresh quotes. Setting appropriate expiry times (neither too short nor too long) balances this risk.

Operator capacity exhaustion: An operator signs multiple quotes simultaneously, each consuming capacity, but cannot fulfill all of them. This is the operator's responsibility to manage. Operators should track outstanding quotes and decline requests when capacity is committed. Operators who overcommit face service quality degradation and potential slashing.

Network partition: If the P2P network partitions, some customers may not reach some operators. Quote collection fails or produces incomplete results. Recovery occurs when the partition heals. The system does not guarantee liveness during network failures; it guarantees safety (committed quotes remain valid).

5.8 Price Discovery and Market Dynamics

The RFQ system enables genuine price discovery through competitive dynamics. Understanding these dynamics helps participants navigate the marketplace effectively.

When a customer needs a service, they can query multiple operators simultaneously. Operators compete on price, reliability, and reputation. Customers select the combination that best meets their needs. This creates healthy market dynamics: operators who price too high lose customers to competitors; operators who price too low may not cover their costs.

Price convergence occurs as operators observe market conditions. On-chain service creation events reveal what prices customers accepted. Operators can analyze historical prices for similar services, adjusting their quotes to remain competitive. Over time, prices converge toward efficient levels reflecting actual resource costs plus reasonable margins.

Quality differentiation prevents pure price competition. Operators with better reliability, faster hardware, stronger security practices, or superior reputation can command premium prices. Customers willing to pay more for quality create market segments. Not all operators need to compete on price; some compete on value.

New operator bootstrapping presents a challenge: operators with no history have no reputation. How do they attract initial customers? Several mechanisms help. Operators can offer introductory pricing below market rates. They can stake additional assets, signaling commitment. Blueprint developers can maintain recommended operator lists that include new entrants. Over time, successful execution builds the reputation that enables market-rate pricing.

Collusion resistance comes from market openness. Any operator meeting registration requirements can participate. A cartel attempting to fix prices faces entry by non-cartel operators who undercut the fixed price. Sustained collusion requires controlling market entry, which the permissionless protocol prevents.

5.9 Operator Discovery

Customers discover operators through multiple channels, each suited to different needs.

On-chain, customers query which operators are registered for their desired blueprint, examining registration timestamps, stake amounts, and historical metrics (jobs completed, success rates, slashing history). This data is public and verifiable.

Off-chain, customers connect to the peer-to-peer gossip network where operators advertise availability, or query operator RPC endpoints directly. The protocol stores operator RPC addresses at registration, enabling direct communication.

Reputation aggregators may emerge as ecosystem infrastructure. These services collect and present operator metrics, ratings, and reviews. They might weight different factors (reliability, price, speed) according to customer preferences. The protocol provides raw data; aggregators provide curation.

Blueprint-specific recommendations come from blueprint developers who understand their service's requirements. A developer might maintain a list of operators known to perform well for their specific blueprint. Customers trust developer judgment as a proxy for operator quality.

The protocol provides primitives; ecosystem participants build discovery mechanisms suited to their needs. This separation allows discovery to evolve independently of protocol upgrades.

6 Fee Distribution and Rewards

Services are now created and priced through the marketplace. The next question is where the money goes. Value flows through Tangle via service fees and inflation rewards, and understanding these flows is essential to understanding participant incentives. This section describes how fees are calculated, split, and distributed to participants.

6.1 Service Fee Splits

When customers pay for services, the payment splits across four recipient categories. The default split allocates 20% to the developer (blueprint owner), 20% to the protocol treasury, 40% to operators (weighted by exposure), and 20% to delegators (weighted by stake).

These percentages are governance-controlled parameters. The community can adjust splits to respond to changing conditions: increasing developer share to incentivize blueprint creation, adjusting operator share to attract more compute providers, or modifying protocol share to fund development priorities.

The split calculation uses floor division for the first three recipients, with the delegator share capturing any rounding dust. This ensures no value is lost to truncation:

$$\begin{align}
\text{developerAmount} &= \lfloor \text{payment} \times \text{developerBps}/10000 \rfloor \tag{1}\\
\text{protocolAmount} &= \lfloor \text{payment} \times \text{protocolBps}/10000 \rfloor \tag{2}\\
\text{operatorAmount} &= \lfloor \text{payment} \times \text{operatorBps}/10000 \rfloor \tag{3}\\
\text{delegatorAmount} &= \text{payment} - \text{developer} - \text{protocol} - \text{operator} \tag{4}
\end{align}$$

6.2 Operator Revenue

The operator allocation distributes among service operators weighted by their committed exposure. An operator with higher exposure receives a proportionally larger share:

$$\text{operatorShare}_i = \frac{\text{operatorAmount} \times e_i}{\sum_j e_j}$$

where e_i is operator i's exposure in basis points.

This creates direct incentive alignment: operators who commit more stake to a service earn more from that service. The relationship is linear and transparent. Operators can calculate their expected revenue from exposure commitments and service pricing.

6.3 Delegator Rewards

The delegator allocation flows through the operator to their delegators. The distribution uses score-based accounting, where a delegator's score depends on both their stake amount and their lock duration.

Lock duration multipliers reward longer commitments:

Lock Duration	Multiplier
None	1.0x
One Month	1.1x
Two Months	1.2x
Three Months	1.3x
Six Months	1.6x

A delegator's score is amount × multiplier. Rewards distribute proportionally to score. This incentivizes stable, long-term delegation over short-term opportunistic staking.

Distribution uses accumulated-per-share accounting for O(1) efficiency. When rewards R arrive for a pool with total score T, the accumulator increases:

$$\text{accumulatedPerShare} \mathrel{+}= \frac{R \times 10^{18}}{T}$$

Delegators claim by computing their share of accumulation since their last claim:

$$\text{claimable} = \frac{\text{score} \times (\text{accumulated} - \text{lastDebt})}{10^{18}}$$

No iteration is required regardless of delegator count.

6.4 Developer Rewards

Developers earn from both service fees (as the developer share of each payment) and from inflation rewards through the developer scoring system.

Developer scoring combines multiple factors:

$$\text{blueprintScore} = \text{blueprintCount} \times 500 \tag{5}$$

$$\text{serviceScore} = \text{serviceCount} \times 1000 \tag{6}$$

$$\text{jobScore} = \text{jobCount} \times 100 \tag{7}$$

$$\text{feeScore} = \sqrt{\text{totalFees}/10^{18}} \times 10^{9} \tag{8}$$

The square root on fees prevents whale dominance. A developer generating \$1M in fees scores the same as ten developers each generating \$10K. This ensures that useful blueprints with broad adoption are rewarded, not just blueprints used by a few large customers.

Service count receives higher weight than blueprint count. This incentivizes developers to create blueprints that achieve adoption rather than many unused blueprints.

6.5 TNT Token Utility

The Tangle Network Token (TNT) serves multiple functions within the ecosystem.

For staking, operators and delegators stake TNT to participate in services and earn rewards. TNT is the primary staking asset, though the protocol supports multi-asset staking with governance-approved tokens.

For governance, TNT holders vote on protocol upgrades, parameter changes, and treasury allocation. Voting power is proportional to stake.

For payments, service fees may be paid in TNT, with optional discounts funded from the protocol's share of fees.

For scoring, TNT can receive a governance-controlled score rate. When enabled, 1 TNT equals a fixed USD score value (for example, \$1) regardless of market price. This allows TNT stakers to earn a predictable share of fee distributions independent of price volatility. The score rate is configurable via governance, allowing the community to adjust incentives as market conditions change.

7 Inflation and Epoch Distribution

Service fees reward participants for current activity, but nascent networks need additional incentives to bootstrap. Beyond service fees, participants earn from inflation rewards distributed by the InflationPool. These rewards provide income even when service demand is still developing, encouraging early participation that builds network capacity. This section describes the inflation model and distribution mechanics.

7.1 The Pre-Funded Model

The InflationPool uses a pre-funded rather than minting model. The pool cannot create new tokens; it only distributes tokens it already holds. Governance or treasury operations fund the pool with yearly inflation allocations.

This design provides risk isolation. If the pool contract were compromised, attackers could steal only the pool's current balance, not mint unlimited tokens. It also enables replaceability: governance can deploy a new pool and migrate funds without modifying token contracts.

The pool calculates each epoch's budget based on remaining balance and remaining time:

$$\text{epochBudget} = \frac{\text{poolBalance}}{\text{epochsRemaining}}$$

This smooths distribution over the funding period. Early epochs do not deplete the pool; later epochs receive comparable budgets.

7.2 Distribution Weights

Inflation distributes across four participant categories with governance-controlled weights. The default allocation is:

Category	Weight
Stakers (delegators)	40%
Operators (performance-based)	25%
Developers (merit-based)	25%
Customers (usage-based)	10%

These weights reflect priorities: incentivizing stake provision for security, rewarding operator performance, encouraging blueprint development, and returning value to active customers. Governance can add additional categories or adjust weights to respond to network needs.

Governance can adjust weights to respond to network needs. If operator supply is insufficient, increasing operator weight attracts more compute. If blueprint quality is lacking, increasing developer weight incentivizes creation. The weights are tools for steering network growth.

7.3 Operator Scoring

Operator inflation rewards distribute based on performance metrics. The scoring formula combines job execution with stake:

$$\text{successRate} = \frac{\text{successfulJobs} \times 10000}{\text{totalJobs}} \tag{9}$$

$$\text{stakeWeight} = \frac{\text{totalStake}}{10^9} \tag{10}$$

$$\text{jobScore} = \frac{\text{jobs} \times \text{successRate} \times \text{stakeWeight}}{10000} \tag{11}$$

$$\text{heartbeatBonus} = \frac{\text{heartbeats} \times \text{stakeWeight}}{100} \tag{12}$$

Stake weight is linear rather than square root. This is an explicit design choice to defeat Sybil attacks. With square root weighting, splitting 100 stake across two operators would yield $\sqrt{50} + \sqrt{50} \approx 14.1$ combined weight versus $\sqrt{100} = 10$ for a single operator. Linear weighting provides no such advantage: $50 + 50 = 100$ regardless of division.

Operators must be staked for a minimum number of epochs (default: one) before earning rewards. This prevents flash-stake attacks where an attacker might borrow stake, claim rewards, and repay within a single transaction.

8 Slashing and Accountability

Fees and inflation provide the carrot; slashing provides the stick. Without credible consequences for misbehavior, the economic security model collapses. An operator who can cheat without penalty will eventually cheat. Slashing ensures that staked assets back service guarantees with genuine economic consequence, transforming stake from mere deposit into true security. This section describes slashing conditions, execution, and protections.

8.1 Developer-Defined Slashing Conditions

The protocol does not prescribe specific slashable offenses. Instead, it provides the slashing mechanism; developers define when and how to use it.

This design reflects the diversity of possible services. An AI sandbox service might slash for using incorrect models, returning results too slowly, or failing to enforce isolation. An oracle service might slash for providing incorrect data. A compute service might slash for job failures or unavailability. Each service type has different requirements; the protocol cannot anticipate them all.

Developers implement slashing conditions through the hook system. The `querySlashingOrigin` hook specifies which address may propose slashes. The `onSlash` hook executes when slashing is finalized. Custom verification contracts can implement arbitration, voting, or cryptographic proof validation.

This expressiveness comes with responsibility. Developers must design effective detection mechanisms. Operators should evaluate slashing conditions before registering for blueprints. The protocol provides infrastructure; participants provide judgment.

8.2 The Dispute Window

Slash proposals do not execute immediately. A dispute window (default seven days) provides operators opportunity to contest accusations.

The lifecycle proceeds through phases. In the proposal phase, an authorized address submits evidence of misbehavior. The protocol computes the effective slash amount, creates a pending proposal, and sets the execution timestamp to current time plus dispute window.

During the dispute period, the accused operator may submit counter-evidence. Protocol administrators may cancel the slash if evidence proves fraudulent. Parties investigate and gather evidence.

After the dispute window expires, any address may execute pending slashes that were not disputed. The protocol validates status and timing, reduces operator stake by the computed amount, affects delegator pools via exchange rate, and marks the slash as executed. Disputed slashes require administrator resolution and cannot be executed automatically.

This structure balances accountability with fairness. Operators face real consequences for misbehavior, but have recourse against false accusations. The dispute window length is governance-controlled, allowing the community to adjust based on experience.

8.3 Proportional Slashing

Slashing is proportional to the operator's committed exposure:

$$\text{effectiveSlashBps} = \frac{\text{slashBps} \times \text{exposureBps}}{10000}$$

An operator with 10% exposure to a service faces at most 10% of their stake at risk for that service's slashing events, regardless of the base slash severity. This bounds downside and enables participation in multiple services without unlimited risk accumulation.

8.4 O(1) Share-Based Execution

When slashing delegator pools, only total assets change:

$$T_a' = T_a - \text{slashAmount}$$

Shares remain constant. The exchange rate T_a/T_s automatically decreases, affecting all delegators proportionally. No per-delegator iteration is required. This enables slashing millions of delegators in a single storage write.

8.5 Withdrawal Protection

The pending slash counter blocks delegator withdrawals while slashes are pending. Without this protection, delegators who learn of impending slashes could withdraw before execution, leaving insufficient stake to cover the penalty.

While any slash proposal is pending against an operator, their delegators cannot withdraw. The counter increments on proposal and decrements on execution or cancellation. Withdrawals resume when no proposals remain pending.

9 Governance

The mechanisms described so far (staking, fees, slashing) are parameters. Who sets those parameters? In centralized systems, platform operators decide unilaterally. In Tangle, decisions belong to stakeholders. Decentralized governance enables protocol evolution without centralized control. Token holders propose changes, vote on referenda, and control protocol parameters. This section describes the governance mechanism, its security properties, and the philosophy guiding its design.

9.1 Governance Philosophy

Protocol governance must balance competing concerns. Flexibility enables adaptation to changing conditions. Stability protects participants who committed based on current rules. Speed enables rapid response to threats. Deliberation prevents hasty mistakes. These tensions cannot be fully resolved; governance design chooses where to strike the balance.

Tangle's governance philosophy prioritizes **safety over liveness**. A proposal that fails to pass does less damage than a malicious proposal that passes. Therefore, governance thresholds are set relatively high, quorum requirements ensure broad participation, and timelocks provide escape hatches for egregious errors. The cost of this philosophy is slower adaptation; the benefit is protection against capture and manipulation.

The philosophy also prioritizes **stake-weighted voting over plutocratic voting**. Token holders vote in proportion to their holdings, reflecting the principle that those with more at stake should have more say in decisions affecting that stake. This is not democratic in the one-person-one-vote sense; it is democratic in the shareholder sense, where economic exposure confers governance rights.

Finally, the philosophy embraces **on-chain execution with off-chain deliberation**. Formal votes occur on-chain with cryptographic finality. Discussion, debate, and consensus-building occur off-chain through forums, calls, and informal coordination. The on-chain mechanism records decisions; the off-chain process generates them.

9.2 Token Governance Mechanism

On-chain governance uses the OpenZeppelin Governor framework with ERC20Votes for stake-weighted voting. The Governor contract implements a complete proposal lifecycle from creation through execution.

Proposal creation requires holding tokens above the proposal threshold. This prevents spam proposals while ensuring that significant stakeholders can always propose changes. The threshold is set to 1% of circulating supply by default, meaning any holder with 1% can unilaterally create proposals.

Voting delay is the period between proposal creation and voting start. During this period, voters can review the proposal, discuss implications, and prepare their positions. The default delay is 2 days, providing reasonable review time without excessive delay.

Voting period is the duration during which votes are accepted. Longer periods enable broader participation (more voters have time to vote) but delay execution. The default period is 7 days, balancing participation against timeliness.

Quorum is the minimum participation required for a valid outcome. Without quorum, a proposal fails regardless of vote distribution. Quorum prevents a small minority from passing proposals when most stakeholders are absent. The default quorum is 10% of circulating supply.

Vote counting uses simple majority: a proposal passes if more tokens vote for than against, assuming quorum is met. Abstentions count toward quorum but not toward passage.

Timelock execution interposes a delay between proposal passage and execution. During this delay, affected parties can prepare for the change or, in extreme cases, exit the protocol. The default timelock is 3 days. For upgrades and critical parameter changes, a longer timelock of 7 days applies.

These parameters themselves are governable. The community can adjust thresholds, delays, and quorum requirements through the standard proposal process, though changes to governance parameters typically require super-majority support.

9.2.1 Parameter Selection Rationale

The default parameters were selected through analysis of existing governance systems and simulation of attack scenarios.

Proposal threshold (1%) balances accessibility against spam. Compound uses 1%, Uniswap uses 2.5%, Aave uses 0.5%. Tangle chose 1% because simulations show this prevents spam (acquiring 1% of supply is expensive) while allowing minority proposals (active community members can realistically accumulate 1%). With expected token distribution, approximately 20-50 addresses should exceed this threshold, ensuring diverse proposal sources.

Quorum (10%) ensures meaningful participation without requiring unrealistic turnout. Historical data from major DAOs shows typical participation of 5-15% on routine proposals, with controversial proposals drawing 20-40%. A 10% quorum requires engagement above baseline apathy while remaining achievable for important decisions. With expected participation patterns (15% average turnout, higher for controversial proposals), this quorum balances the risk of minority capture (too low) against governance paralysis (too high). Governance monitoring should track actual participation rates and adjust quorum if sustained participation diverges significantly from expectations.

Voting period (7 days) provides adequate deliberation time across global time zones. Analysis of governance attacks shows most successful defenses mobilize within 72-96 hours of detection. A 7-day window provides 4-5 days for defensive response after initial detection. Shorter windows (Compound's 3 days) have shown vulnerability to weekend timing attacks.

Timelock (3/7 days) creates escape windows for affected participants. The 3-day standard timelock allows users approximately 100,000 blocks to exit positions if they disagree with passed proposals. The 7-day upgrade timelock provides additional buffer for the most consequential changes, matching the dispute window in slashing to ensure consistency across protocol mechanisms.

9.2.2 Worked Example: Fee Parameter Change

Consider a community proposal to increase the protocol fee from 10% to 15%. The following timeline illustrates the complete governance lifecycle.

Day 0 (Proposal Creation): A token holder with 1.2% of supply creates an on-chain proposal targeting `setProtocolFee(1500)`. The proposal includes a description explaining the rationale (increased protocol revenue for development funding) and links to forum discussion. Transaction cost: approximately 200,000 gas.

Days 0-2 (Voting Delay): The community reviews the proposal. Analysis is posted on the governance forum showing that the fee increase would generate an additional $500,000 annually for the treasury at current volumes, while reducing operator margins by 5%. Some operators post objections; supporters argue the treasury needs funding for security audits.

Days 2-9 (Voting Period): Voting opens. Major delegates announce their positions. By day 5, the proposal shows 8% participation with 65% in favor. An operator coalition mounts opposition, increasing participation to 14% by day 8 with the margin narrowing to 55% in favor.

Day 9 (Voting Ends): Final tally: 14.2% participation (above 10% quorum), 54% in favor, 46% against. The proposal passes by simple majority.

Days 9-12 (Timelock): The 3-day timelock begins. Operators who strongly oppose the change have time to communicate with customers, adjust pricing, or in extreme cases, begin exit procedures. One operator with 3% of staked assets announces they will leave; the community discusses whether this exodus justifies reconsideration, but no on-chain action is taken.

Day 12 (Execution): Anyone can call `execute()` on the proposal. The fee parameter updates to 15%. The operator who announced departure begins their exit queue. Protocol operations continue normally with the new fee.

This example illustrates that governance is not merely voting but a multi-phase process with deliberation, response, and adaptation at each stage.

9.3 Governance Attack Resistance

Decentralized governance creates attack surfaces that centralized systems avoid. Understanding these attacks and their mitigations is essential.

Governance capture occurs when an attacker accumulates enough tokens to pass proposals unilaterally. With a 10% quorum and simple majority, an attacker controlling just over 10% of supply could pass proposals if other voters abstain. Defense relies on the assumption that attacks will trigger defensive participation: when the community perceives a capture attempt, stakeholders vote against. The timelock provides time for this defensive mobilization.

Flash loan attacks attempt to borrow tokens, vote, and return tokens within a single transaction. OpenZeppelin's ERC20Votes implementation defeats this attack through vote checkpointing: voting power is determined by token holdings at a snapshot block, not current holdings. An attacker who borrows tokens after the snapshot gains no voting power.

Whale manipulation involves large holders exercising disproportionate influence. This is inherent to stake-weighted voting and is considered a feature, not a bug: those with more at stake should have more influence. However, whales who abuse their power face social consequences (community opposition, reputation damage) and economic consequences (their stake's value depends on protocol health).

Voter apathy enables minority rule when most token holders do not vote. Quorum requirements and delegation (allowing passive holders to delegate voting power) provide complementary defenses.

Bribery attacks pay voters to vote a particular way. Defense relies on voters' long-term incentives: accepting bribes to pass harmful proposals damages their own holdings, making large-scale bribery expensive.

9.4 Vote Delegation

Token holders may delegate their voting power to other addresses. Delegation enables passive holders to participate in governance through representatives they trust, improving participation without requiring every holder to evaluate every proposal.

Delegation is transitive: if Alice delegates to Bob, and Bob delegates to Carol, Alice's voting power accrues to Carol. Delegation can be changed at any time; the current delegation at the vote snapshot determines power allocation.

Delegation creates a representative layer where informed, engaged community members accumulate voting power from less engaged holders. This can improve governance quality (decisions made by those who understand them) or create centralization risks (power concentrating in few delegates). The community must monitor delegation patterns and consider limits if concentration becomes problematic.

9.5 Emergency Mechanisms

Not all threats can wait for standard governance processes. A critical vulnerability might require immediate response. The protocol includes emergency mechanisms for such situations.

Pause functionality allows designated PAUSER_ROLE holders to halt protocol operations. Pausing prevents new services, suspends job processing, and blocks withdrawals. Pausing does not resolve the underlying issue; it prevents further damage while resolution is developed. Unpausing requires governance approval or ADMIN_ROLE action.

Emergency withdrawal enables the DEFAULT_ADMIN_ROLE to withdraw funds from specific contracts (such as the InflationPool) to a designated safe address. This is a nuclear option for situations where contract bugs threaten fund safety. Emergency withdrawal is logged and auditable; misuse would be immediately visible.

Guardian multisig may hold emergency roles during the protocol's early period. The multisig can act faster than governance in genuine emergencies. Over time, as the protocol matures and governance proves reliable, guardian powers can be revoked or transferred to the timelock.

These mechanisms create centralization risk: the parties holding emergency powers could abuse them. The mitigation is transparency (all actions are on-chain and visible) and social accountability (abuse would be detected and punished through reputation damage and potential legal consequences).

9.6 Governable Parameters

Governance controls a comprehensive set of protocol parameters, enabling adaptation without contract upgrades.

Fee parameters include developer, protocol, operator, and staker fee percentages (default: 20/20/40/20); TNT payment discount rates; and minimum fees for various operations.

Inflation parameters include distribution weights across staking, operators, customers, developers, and restakers (default: 40/25/25/10/0); epoch length (default: 7 days); funding period duration (default: 365 days); and minimum stake epochs before reward eligibility (default: 1).

Staking parameters include minimum operator stakes per asset; delegation mode restrictions; lock duration multipliers (default: 1.0/1.1/1.2/1.3/1.6x); and maximum blueprints per operator.

Slashing parameters include default dispute window duration (default: 7 days); maximum slash percentages (default: 100%); and instant slash enablement (default: disabled).

Service parameters include minimum and maximum TTLs; request expiry grace periods; maximum quote age (default: 1 hour); and heartbeat requirements.

TNT parameters include the TNT score rate for fee distribution and token address configuration for multi-chain deployments.

This extensive parameter set enables governance to tune the protocol in response to market conditions, community feedback, and observed behavior, without requiring contract upgrades for routine adjustments.

9.7 Upgrade Mechanism

For changes beyond parameter adjustment, contract upgrades use the UUPS (Universal Upgradeable Proxy Standard) pattern. Each upgradeable contract (Tangle, MultiAssetDelegation, InflationPool, etc.) is deployed behind a proxy. The proxy delegates calls to an implementation contract; upgrading replaces the implementation while preserving storage.

The upgrade authorization function is protected by the UPGRADER_ROLE, which is held by the timelock. Upgrades therefore follow the standard governance process: proposal, voting, timelock delay, execution. This ensures that significant changes receive community review.

UUPS was chosen over transparent proxy and diamond patterns for several reasons. UUPS places upgrade logic in the implementation rather than the proxy, reducing proxy size and attack surface. UUPS allows upgrades to disable upgradeability entirely if desired (by not including upgrade functions in the new implementation). UUPS is well-audited through OpenZeppelin's widely-used implementation.

Upgrades are the most powerful governance action. A malicious upgrade could drain funds, disable slashing, or arbitrary modify behavior. Therefore, upgrade proposals receive the longest timelock (7 days) and highest scrutiny. The community should treat every upgrade proposal as potentially dangerous until proven otherwise.

9.8 Off-Chain Governance

On-chain voting is the final decision mechanism, but off-chain processes inform formal votes: forum discussion for proposal drafting, temperature checks for gauging sentiment, working groups for domain expertise, and improvement proposals following structured formats. The on-chain mechanism provides finality; the off-chain process provides deliberation.

10 Security Model

The preceding sections described individual mechanisms: how staking works, how fees flow, how slashing executes, how governance operates. This section synthesizes these mechanisms into a coherent security analysis, presenting Tangle's security model: the threats the protocol defends against, the mechanisms providing defense, and the formal properties these mechanisms guarantee. Security in a cryptoeconomic protocol differs fundamentally from traditional security: the goal is not to make attacks impossible but to make them unprofitable.

10.1 Threat Model and Adversaries

Tangle's security analysis considers three adversary classes with increasing capabilities.

Rational adversaries are profit-motivated actors who will attack if and only if the expected value of attacking exceeds the expected cost. This is the primary adversary model. Rational adversaries will not attack systems where the expected value is negative, even if the attack is technically feasible. The defense against rational adversaries is economic: ensure that for any attack, the expected slashing cost exceeds the expected gain.

Formally, let V be the value an adversary gains from a successful attack, p_d be the probability of detection, and S be the slashing amount. A rational adversary attacks if and only if:

$$V > p_d \cdot S$$

The protocol's security condition is therefore:

$$p_d \cdot S \geq V$$

Blueprints satisfy this condition by configuring appropriate exposure requirements (which determine S) and verification mechanisms (which determine p_d).

Byzantine adversaries are adversaries who may behave arbitrarily, including irrationally. A Byzantine adversary might attack even when the expected value is negative—motivated by ideology, error, or external incentives invisible to the protocol. Defense against Byzantine adversaries requires bounding the damage any single party can cause.

The protocol bounds Byzantine damage through several mechanisms. Stake requirements limit operator participation to those with economic resources at risk. Exposure limits cap how much stake backs any single service. Dispute windows provide time for detection and response. Isolation mechanisms contain damage to the affected service.

Colluding adversaries are groups of adversaries who coordinate to overcome defenses designed for individual actors. If k operators collude, they may control sufficient stake to extract value exceeding their combined slashing cost, or they may evade detection mechanisms that rely on independent verification.

Defense against collusion requires structural mechanisms. Randomized operator selection prevents adversaries from guaranteeing collusion partners serve the same service. Stake distribution requirements ensure no single entity controls excessive stake. Cross-blueprint verification enables independent parties to check each other's work.

10.2 Economic Security Analysis

Economic security quantifies how much an adversary must spend to successfully attack the protocol. Higher economic security deters more rational adversaries.

10.2.1 Cost of Corruption

The cost of corruption (CoC) is the minimum amount an adversary must control or acquire to corrupt a service. For a service with n operators each staking s_i with exposure e_i, the total stake at risk is:

$$\text{StakeAtRisk} = \sum_{i=1}^{n} s_i \cdot e_i$$

If successful corruption requires controlling k of n operators (e.g., for a majority-vote service), the cost of corruption is the minimum stake among any k operators:

$$\text{CoC} = \min_{\substack{T \subseteq \{1,\ldots,n\} \\ |T|=k}} \sum_{i \in T} s_i \cdot e_i$$

Customers select operators to maximize CoC given their security requirements. The RFQ mechanism enables customers to specify minimum stake requirements, and operators quote based on meeting those requirements.

10.2.2 Profit from Corruption

The profit from corruption (PfC) is the maximum value an adversary can extract from corrupting a service. This depends on what the service does. An oracle service reporting asset prices has PfC equal to the maximum arbitrage profit from false prices. A trading agent service has PfC equal to the assets under management. A data processing service has PfC equal to the value of data confidentiality.

Blueprints must analyze their PfC and configure exposure requirements accordingly. A service with PfC of \$1 million requires operators to stake at least \$1 million in exposed assets. A service with PfC of \$10 million requires at least \$10 million.

10.2.3 Security Ratio

The security ratio is CoC/PfC. A ratio above 1 means honest behavior is more profitable than corruption under perfect detection. A ratio below 1 means corruption is potentially profitable.

$$\text{SecurityRatio} = \frac{\text{CoC}}{\text{PfC}}$$

The protocol does not enforce minimum security ratios—blueprints determine their own requirements based on their threat model. However, we recommend a **minimum security ratio of 1.5x** for production services. This buffer accounts for imperfect detection (real detection probability $< 100\%$), token price volatility (stake value may decline during the dispute window), and adversary risk tolerance (attackers may accept lower expected value for high-variance outcomes). High-value services managing significant assets should target ratios of 2x or higher.

10.2.4 Worked Example: Trading Agent Service

Consider a trading agent service managing \$500,000 in customer assets. The service uses 3-of-5 majority voting, meaning an adversary must corrupt at least 3 operators to manipulate trading decisions. The five operators have the following stake and exposure configurations:

Operator	Stake	Exposure	Stake at Risk
A	$400,000	50%	$200,000
B	$300,000	60%	$180,000
C	$250,000	80%	$200,000
D	$200,000	100%	$200,000
E	$150,000	100%	$150,000

The cost of corruption is the minimum stake at risk among any 3 operators. The cheapest 3-operator coalition is {B, D, E} with combined stake at risk of $180,000 + $200,000 + $150,000 = $530,000.

The profit from corruption equals the assets under management: PfC = $500,000.

The security ratio is:

$$\text{SecurityRatio} = \frac{\$530,000}{\$500,000} = 1.06$$

A security ratio above 1 means the expected loss from slashing exceeds the potential gain under perfect detection. However, the simple threshold of 1 is insufficient for production systems. Detection probability is rarely 100%; token prices fluctuate; and rational adversaries apply risk discounts. Industry practice and academic analysis suggest a **minimum security ratio of 1.5x**, with 2x or higher preferred for high-value services.

In this example, the 1.06 ratio provides only 6% margin. A 10% token price drop yields a security ratio of 0.95, making attack profitable. Even without price movement, if detection probability is 90% rather than 100%, the expected cost of corruption becomes only $0.9 \times \$530,000 = \$477,000$, making the attack marginally profitable.

Recommendation: This service should increase its security ratio to at least 1.5x through: (a) increasing operator exposure requirements to 75-100%, (b) adding additional operators to increase the corruption threshold to 4-of-7 or higher, (c) requiring operators to stake at least $400,000 each, or (d) implementing monitoring to pause operations if token prices decline more than 20% from baseline.

10.2.5 Flash Loan Resistance

Flash loans enable adversaries to temporarily acquire large amounts of capital without economic exposure. An adversary could borrow tokens, stake them, participate in a service, attack, and repay the loan—all in a single transaction. This would reduce the cost of corruption to the loan fee rather than the stake value.

Tangle prevents flash loan attacks through temporal separation. Stake must remain locked for a minimum number of epochs before operators can participate in services. The `minStakeEpochs` parameter (default: 1 epoch, approximately 1 week) ensures that staked assets remain at risk for meaningful duration. Flash loans, which must be repaid within a single transaction, cannot satisfy this requirement.

Additionally, slashing proposals must survive a dispute window (default: 7 days) before execution. Even if an adversary somehow participated in a service, slashing would execute long after any flash loan was repaid.

10.3 Attack Vectors and Mitigations

This section analyzes specific attack vectors and the mechanisms defending against them.

10.3.1 Withdrawal Front-running

An operator who knows they will be slashed might attempt to withdraw their stake before the slash executes. If successful, slashing would have no effect, and the security model would collapse.

Mitigation: Pending slash proposals block withdrawals. When a slash is proposed, the protocol increments the operator's `pendingSlashCount`. Withdrawal functions check this counter and revert if non-zero. The counter decrements only when the slash is executed or cancelled. An operator under slash proposal cannot withdraw until the dispute resolves.

10.3.2 Stake Manipulation

An adversary might attempt to manipulate stake accounting to increase their share of rewards or reduce their slash exposure. Share-based accounting systems can be vulnerable to donation attacks (inflating the exchange rate) or first-depositor attacks (extracting value from subsequent depositors).

Mitigation: Virtual offset prevents first-depositor attacks. The share calculation includes a virtual offset V:

$$\text{shares} = \text{amount} \times \frac{T_s + V}{T_a + V}$$

This offset ensures that even the first depositor receives shares at a reasonable exchange rate, preventing extraction from subsequent depositors.

Donation attacks (depositing assets directly to the pool without minting shares) would increase the exchange rate, benefiting existing shareholders. The protocol accepts this as benign—donors lose value, existing stakers gain value. Malicious donation would be irrational.

10.3.3 Slashing Griefing

An adversary might propose slashes against innocent operators, forcing them to spend resources on dispute resolution and damaging their reputation. Repeated griefing could discourage operator participation.

Mitigation: Slash proposal requires governance approval or comes from the blueprint's service manager. Arbitrary addresses cannot propose slashes. Blueprints that allow open slash proposals must implement their own griefing protections (e.g., requiring slasher bonds that are forfeited if the slash is cancelled).

10.3.4 Service Denial

An adversary might request services with no intention of using them, consuming operator capacity and preventing legitimate customers from accessing services.

Mitigation: Payment accompanies service requests. For pay-once services, the full payment is collected at request time. For subscription services, customers must fund escrow. For event-driven services, jobs include payment. An adversary denying service must pay for the denial, making sustained attacks expensive.

10.3.5 Operator Collusion

Operators serving the same service might collude to provide false results, split the profits, and avoid detection by presenting consistent (but incorrect) outputs.

Mitigation: Collusion resistance depends on blueprint design. Services requiring Byzantine fault tolerance should use verification mechanisms that detect inconsistency (redundant execution, cryptographic attestation). Services requiring confidentiality should use trusted execution environments. The protocol provides the framework; blueprints implement collusion resistance appropriate to their threat model.

Structural mitigations include stake distribution requirements (limiting concentration), randomized operator assignment (preventing adversaries from guaranteeing colluding partners), and cross-service verification (independent parties checking each other).

10.3.6 Governance Capture

An adversary acquiring sufficient voting power might pass proposals benefiting themselves at the expense of the protocol—reducing slashing parameters, increasing their reward share, or modifying contracts to extract funds.

Mitigation: Multiple layers protect against governance capture. High proposal thresholds (1% of supply) prevent low-stake capture. Long voting periods (7 days) allow community response. Timelock delays (3-7 days) enable exit before harmful proposals execute. Governance cannot modify core invariants without protocol upgrades, which require UUPS authorization and higher approval thresholds.

10.4 Formal Security Properties

This section states formal properties that Tangle's mechanisms guarantee.

Property 1 (Withdrawal Safety): An operator with a pending slash proposal cannot reduce their stake below the slash amount until the proposal resolves.

Mechanism: The `pendingSlashCount` blocks withdrawals when non-zero. Slash proposals increment this counter; execution or cancellation decrements it.

Property 2 (Proportional Slashing): When a slash of $x\%$ executes against an operator, all delegators to that operator lose exactly $x\%$ of their delegation value.

Mechanism: Share-based accounting adjusts `totalAssets` without modifying `totalShares`. The exchange rate $r = T_a/T_s$ decreases by exactly $x\%$, reducing all positions proportionally.

Property 3 (Dust Conservation): Payment splits distribute exactly the input amount across recipients, with no dust lost to rounding.

Mechanism: The final recipient receives the remainder after all other recipients receive floored amounts: final = total $-\sum$ others.

Property 4 (Temporal Separation): Stake cannot participate in services until it has been locked for at least `minStakeEpochs`.

Mechanism: Operator registration records the stake epoch. Service approval validates that current epoch exceeds registration epoch by `minStakeEpochs`.

Property 5 (Bounded Governance): Governance proposals cannot execute until timelock delay has elapsed, and emergency mechanisms require multi-party authorization.

Mechanism: `TangleTimelock` enforces minimum delay. Emergency functions require guardian multisig signatures in addition to governance approval.

10.5 Security Assumptions

Tangle's security relies on assumptions that, if violated, would compromise the guarantees above.

Assumption 1 (Rational Majority): The majority of stake is controlled by rational adversaries who will not attack when the expected value is negative.

If a majority of stake is controlled by Byzantine adversaries willing to accept losses, they could corrupt services, execute governance attacks, or manipulate slashing.

Assumption 2 (Cryptographic Hardness): The cryptographic primitives (ECDSA signatures, BLS signatures, hash functions, Merkle proofs) remain secure against computational attacks.

Advances in cryptanalysis or quantum computing could compromise signature schemes, enabling forgery of operator approvals, slash proposals, or governance votes.

Assumption 3 (Smart Contract Correctness): The deployed smart contracts correctly implement the specified behavior.

Bugs, vulnerabilities, or undiscovered edge cases could enable attacks not analyzed in the threat model. Audits, formal verification, and bug bounties mitigate but do not eliminate this risk.

Assumption 4 (Economic Stability): Token prices remain stable enough that slashing costs maintain their deterrent value.

Extreme price volatility could reduce slashing costs below profitable attack thresholds. Services with high PfC should monitor token prices and adjust exposure requirements accordingly.

10.6 Security Recommendations

Based on this analysis, blueprints should follow these security recommendations:

Size stake requirements appropriately. The minimum stake should exceed the maximum profit from corruption. Services managing \$X should require operators to stake at least \$X with appropriate exposure.

Implement detection mechanisms. Slashing is only effective if misbehavior is detected. Blueprints should implement redundant execution, cryptographic attestation, or other verification appropriate to their threat model.

Use multi-operator services for high-value applications. Single-operator services have a single point of failure. Multi-operator services with appropriate aggregation (majority voting, threshold signatures) require collusion to corrupt.

Monitor economic conditions. If token prices decline significantly, the cost of corruption declines proportionally. High-value services should monitor conditions and potentially pause operations if security margins become inadequate.

Participate in governance. Governance capture is a systemic risk. Stakeholders who do not participate in governance delegate their protection to others who may not share their interests.

11 Comparison to Existing Restaking Protocols

Tangle does not exist in isolation. Several restaking protocols have emerged, each with different design goals, architecture choices, and target applications. Understanding how Tangle relates to these alternatives helps contextualize its contributions and clarify when each approach is appropriate.

11.1 The Restaking Landscape

Restaking protocols share a common premise: staked assets securing one system can simultaneously secure additional systems, providing economic security without requiring each system to bootstrap its own stake from scratch. This shared security model reduces the capital required to secure new protocols and creates yield opportunities for stakers.

EigenLayer pioneered mainstream restaking on Ethereum, enabling ETH stakers to opt into additional slashing conditions in exchange for additional yield. EigenLayer's AVS (Actively Validated Services) model allows protocols to define custom slashing conditions while drawing security from restaked ETH. The protocol focuses primarily on validators who already stake ETH in the Beacon Chain, extending their capital efficiency.

Symbiotic emphasizes flexibility in collateral types and modular design. Unlike EigenLayer's initial focus on ETH, Symbiotic accepts diverse collateral including LSTs, stablecoins, and other ERC-20 tokens from launch. The protocol separates concerns more granularly, with distinct components for vaults, operators, and networks.

Karak introduces a risk-adjusted approach, allowing different collateral types with different risk profiles to secure different service types. The protocol emphasizes matching collateral risk to service requirements rather than treating all stake uniformly.

11.2 Differentiating Tangle

Tangle differs from these protocols in several fundamental ways.

Focus on compute services rather than validation. Existing restaking protocols primarily secure validation tasks: attestation, ordering, data availability. Tangle focuses on compute services: AI inference, data processing, orchestration, and general computation. This focus drives architectural decisions throughout the protocol. Blueprints define arbitrary computation, not just validation rules. Jobs represent work to be performed, not messages to be validated. The sandbox runtime provides execution environments for computational workloads.

Developer-defined service types. EigenLayer's AVS model and similar constructs require services to integrate with the protocol at a deep technical level. Tangle's blueprint system enables developers to define new service types through configuration and hooks rather than protocol modification. A developer can create a new service category (AI inference, oracle, keeper) without protocol upgrades, using the hook system to specify custom validation, pricing, and slashing behavior.

Integrated products rather than pure infrastructure. Existing restaking protocols provide infrastructure that other projects build upon. Tangle combines infrastructure (the protocol) with products (the sandbox runtime, the workbench) that demonstrate and utilize that infrastructure. This vertical integration provides both a proof of concept and a revenue-generating application from launch.

O(1) algorithms as fundamental design. While other protocols face scalability challenges as participant counts grow, Tangle's share-based accounting and accumulated-per-share rewards provide O(1) operations regardless of participant count. This is not an optimization but a fundamental design choice that shapes the protocol's architecture.

RFQ-based service marketplace. Rather than fixed pricing or AMM-based mechanisms, Tangle uses request-for-quote for service pricing. This choice reflects the heterogeneous nature of compute services: different operators have different capabilities, and customers have different requirements. RFQ accommodates this heterogeneity naturally.

11.3 When to Choose Each Approach

Different protocols suit different needs.

Choose EigenLayer when securing validation services that require Ethereum-aligned security, when the primary need is extending existing ETH stake to new purposes, or when building services closely coupled to Ethereum consensus.

Choose Symbiotic when requiring maximum flexibility in collateral types, when building modular systems that compose various protocol components, or when diverse asset backing is important.

Choose Karak when risk-adjusted collateral matching is the primary concern, when different services require different risk profiles, or when the focus is on optimizing capital efficiency across risk categories.

Choose Tangle when building compute-intensive services (AI, data processing, orchestration), when developer expressiveness and custom service types are important, when the use case benefits from integrated products (sandbox runtime, workbench), or when O(1) scalability to millions of participants is required.

These protocols may also compose. An operator might restake ETH through EigenLayer for validation services while participating in Tangle for compute services. The restaking ecosystem is not zero-sum; different protocols serve different niches within the broader shared security paradigm.

11.4 Future Evolution

The restaking landscape will continue evolving. Protocols will expand capabilities, enter each other's domains, and potentially interoperate. Tangle's position focuses on compute services and developer tooling, domains where existing protocols have less coverage and where AI infrastructure demands will grow significantly.

As AI agents become more prevalent and their computational demands increase, the need for decentralized compute infrastructure will grow correspondingly. Tangle is positioned to serve this demand, with architecture optimized for compute services and products demonstrating what decentralized AI infrastructure enables.

Part III

Developer Platform

Part II described the protocol's mechanics: how blueprints, services, operators, fees, and slashing work together. But the protocol is infrastructure, not product. What makes that infrastructure useful is what developers build on top of it.

This part describes the developer platform: the tools, patterns, and capabilities that enable developers to create services. The hook system provides extensibility, allowing blueprints to customize behavior at every lifecycle stage. The Blueprint SDK provides the tooling for building operator software. Verifiability and detection describe the approaches that make economic security meaningful for different service types.

By the end of this part, developers should understand not just how to use the protocol but how to build sophisticated, secure, and economically sound services on top of it.

12 The Hook System

The hook system enables developers to customize blueprint behavior beyond the protocol's defaults. Hooks are callback functions invoked at specific lifecycle stages, enabling custom validation, authorization, and business logic.

12.1 Extensibility Philosophy

The protocol provides infrastructure; developers provide application logic. This separation enables diverse use cases without protocol modification. An AI sandbox service has different requirements than an oracle network, which differs from a keeper system. Rather than encoding all possible behaviors, the protocol exposes hooks that developers override.

Developers implement the `IBlueprintServiceManager` interface, providing callback implementations for the hooks they wish to customize. Default implementations pass through to protocol behavior for hooks developers do not override.

12.2 Lifecycle Hooks

Lifecycle hooks fire at key stages of blueprint and service existence.

`onBlueprintCreated` fires when a new blueprint is registered. Developers use this to initialize state, validate configuration, or emit custom events.

`onRegister` fires when an operator registers for the blueprint. Developers use this to validate operator qualifications, check stake requirements beyond protocol minimums, or collect registration fees.

`onServiceInitialized` fires when a service activates. Developers use this to set up per-service state, notify external systems, or configure monitoring.

`onServiceTermination` fires when a service ends. Developers use this to clean up state, finalize metrics, or trigger settlement logic.

12.3 Slashing Customization

Slashing hooks enable custom detection and authorization:

```
interface IBlueprintServiceManager {
    /// @notice Return the address authorized to propose slashes
    function querySlashingOrigin(uint64 serviceId)
        external view returns (address);

    /// @notice Called during dispute window
    function onUnappliedSlash(
        uint64 serviceId,
        bytes calldata offender,
        uint8 slashPercent
    ) external;

    /// @notice Called when slash is finalized
    function onSlash(
        uint64 serviceId,
        bytes calldata offender,
        uint8 slashPercent
    ) external;
}
```

`querySlashingOrigin` returns the address authorized to propose slashes for a service. By default, this is the service manager contract itself. Developers can override to delegate slashing authority to dispute resolution contracts, DAOs, or other arbitration mechanisms.

`onUnappliedSlash` fires during the dispute window. Developers use this to gather evidence, notify parties, or trigger investigation workflows.

`onSlash` fires when slashing executes. Developers use this to update reputation systems, eject operators from future services, or log metrics.

12.4 Membership Customization

Membership hooks control who may join and leave services:

```
interface IBlueprintServiceManager {
    /// @notice Check if operator may join
    function canJoin(uint64 serviceId, address operator)
        external view returns (bool);

    /// @notice Check if operator may leave
    function canLeave(uint64 serviceId, address operator)
        external view returns (bool);

    /// @notice Configure exit timing
    function getExitConfig(uint64 serviceId) external view returns
       (
        bool useDefault,
        uint64 minCommitmentDuration,
        uint64 exitQueueDuration,
        bool forceExitAllowed
    );
}
```

`canJoin` determines whether an operator may join a dynamic service. Developers can require minimum reputation, specific hardware attestations, or geographic requirements.

`canLeave` determines whether an operator may exit. Developers can prevent exits during critical periods or require minimum service durations.

`getExitConfig` specifies timing for the exit process: minimum commitment duration before exits are allowed, queue duration after scheduling, and whether service owners may force-remove operators.

12.5 Job and Aggregation Configuration

Job hooks configure execution requirements:

```
interface IBlueprintServiceManager {
    /// @notice Number of results required
    function getRequiredResultCount(uint64 serviceId, uint8
       jobIndex)
        external view returns (uint32);

    /// @notice Whether job requires BLS aggregation
    function requiresAggregation(uint64 serviceId, uint8 jobIndex)
        external view returns (bool);

    /// @notice Aggregation threshold configuration
    function getAggregationThreshold(uint64 serviceId, uint8
       jobIndex)
        external view returns (uint16 thresholdBps, uint8
           thresholdType);
}
```

`getRequiredResultCount` specifies how many operators must submit results. Zero means all operators; positive values mean that specific count.

`requiresAggregation` indicates whether jobs should use BLS signature aggregation for multi-operator consensus.

`getAggregationThreshold` specifies what percentage of operators must sign for a valid aggregated result. The threshold type indicates whether to measure by operator count or by stake weight.

13 The Blueprint SDK

The Blueprint SDK provides the tooling for building services that interact with the Tangle protocol. This section describes the SDK's architecture, design rationale, and the path from concept to deployed service.

13.1 Why Rust

The SDK is written in Rust. This choice reflects several considerations that matter for operator software.

Performance is critical for operators processing high-throughput workloads. Rust compiles to native code with zero-cost abstractions, enabling performance comparable to C/C++ without sacrificing safety. Operators can handle thousands of jobs per second without garbage collection pauses or interpreter overhead.

Safety prevents the memory errors and race conditions that plague systems software. Rust's ownership system catches these bugs at compile time. For operator software handling valuable stake and sensitive customer data, safety is non-negotiable. A buffer overflow or data race in operator software could lead to slashing or data breach.

Async runtime (via Tokio) provides efficient concurrent execution. Operators simultaneously handle events from the blockchain, the P2P network, background services, and customer requests. Rust's async ecosystem manages these concurrent workloads efficiently.

WebAssembly compilation enables portable execution. Operators can compile blueprint logic to WebAssembly for sandboxed execution, verification, or cross-platform deployment.

Ecosystem compatibility with the blockchain tooling ecosystem matters. Key libraries (alloy for Ethereum interaction, libp2p for networking, ark-bn254 for BLS signatures) have mature Rust implementations.

Alternative languages were considered. Go offers good concurrency but lacks Rust's memory safety guarantees. TypeScript offers rapid development but insufficient performance for high-throughput services. C++ offers performance but lacks safety guarantees. Rust provides the best combination for operator software requirements.

13.2 Architecture Overview

The SDK is organized into components that compose into complete operator services. Understanding this architecture helps developers navigate the SDK and design their blueprints effectively.

Producers generate job calls from external events. The `TangleEvmProducer` watches blockchain events. The `CronProducer` generates calls on schedules. Custom producers can watch APIs, message queues, or any other event source.

Router dispatches job calls to appropriate handlers. When a job call arrives, the router examines the job index and invokes the corresponding handler function. This pattern separates routing logic from business logic.

Handlers implement job-specific logic. Each handler receives job context (extractors) and returns a result. Handlers are async functions that can perform computation, call external APIs, interact with databases, or any other operation needed to produce results.

Consumer submits results to the blockchain. For individual results, the consumer calls the contract directly. For aggregated results, the consumer coordinates with other operators to collect signatures before submission.

Background services run alongside the main job processing loop. The SDK provides two abstractions for background work:

The `BackgroundService` trait is the general abstraction for any long-running task. It defines a single `start()` method that spawns a background task and returns a channel for completion notification. This suits servers, monitors, synchronization loops, or any continuous operation. Background services are added to the runner via `.background_service(service)`.

The `BackgroundKeeper` trait is specialized for Tangle lifecycle automation. It defines `start()` for spawning and `check_and_execute()` for single iterations. The SDK provides built-in keepers: `EpochKeeper` for inflation distribution, `RoundKeeper` for round-based protocols, and `StreamKeeper` for payment stream settlement. Developers extend this trait for custom lifecycle monitoring.

P2P layer provides communication between operators. Quote dissemination, signature aggregation, and custom coordination protocols use this layer.

The execution flow proceeds as follows. A producer detects an event (blockchain event, cron trigger, or custom condition) and generates a job call. The router dispatches the call to the appropriate handler. The handler executes and returns a result. The consumer submits the result (coordinating aggregation if needed). Throughout this flow, background services continue running, and the P2P layer handles coordination messages.

13.3 A Minimal Blueprint Example

Understanding the SDK is easier with a concrete example. Consider a minimal blueprint that squares numbers submitted as jobs.

The blueprint defines one job type: `square`. Customers submit a number; operators return its square. The service requires a single operator and uses direct (non-aggregated) result submission.

```rust
use blueprint_sdk::tangle_evm::extract::{TangleEvmArg,
    TangleEvmResult};
use blueprint_sdk::tangle_evm::{TangleEvmConsumer,
    TangleEvmProducer, TangleEvmLayer};
use blueprint_sdk::{Router, runner::BlueprintRunner};

```

```
pub const SQUARE_JOB: u8 = 0;

// Job function: extracts input, returns wrapped result
pub async fn square(TangleEvmArg(x): TangleEvmArg<u64>) ->
    TangleEvmResult<u64> {
     TangleEvmResult(x * x)
}

pub fn router() -> Router {
     Router::new().route(SQUARE_JOB, square.layer(TangleEvmLayer))
}

#[tokio::main]
async fn main() -> Result<(), blueprint_sdk::Error> {
     let env = BlueprintEnvironment::load()?;
     let client = env.tangle_evm_client().await?;
     let service_id = env.protocol_settings.tangle_evm()?.service_id
         .unwrap();

     BlueprintRunner::builder(TangleEvmConfig::default(), env)
          .router(router())
          .producer(TangleEvmProducer::new(client.clone(), service_id
             ))
          .consumer(TangleEvmConsumer::new(client))
          .run()
          .await
}
```

The SDK uses an extractor pattern inspired by web frameworks. `TangleEvmArg<T>` extracts ABI-encoded inputs from job calls. `TangleEvmResult<T>` wraps outputs for submission. Job functions are plain async functions. The `TangleEvmLayer` middleware handles encoding and decoding. The main function composes SDK components: a Tangle event producer watches for job submissions, a router directs jobs to the handler, and a consumer submits results back to the contract.

This minimal example omits error handling, configuration, and operational concerns, but demonstrates the core pattern. Real blueprints add complexity incrementally: multiple job types, aggregation, background services, verification logic.

13.4 Triggers in Depth

Triggers initiate job execution. The SDK provides multiple trigger types for different use cases.

Event triggers watch for on-chain events. The `TangleEvmProducer` subscribes to Tangle contract events via WebSocket, filters for the configured service, extracts job parameters, and produces job calls. Configuration specifies the RPC endpoint, contract address, service ID, and event filters. The producer handles reconnection, block reorganizations, and missed events during downtime.

Cron triggers execute on time-based schedules. The `CronJob` producer implements the `Stream` trait, generating job calls on cron schedules. It accepts standard cron expressions with seconds precision (six fields: second, minute, hour, day, month, weekday). Example: `"* * * * * *"` triggers every second; `"0 */5 * * * *"` triggers every 5 minutes. Cron triggers suit periodic tasks like heartbeats, maintenance, report generation, or polling external APIs. Multiple producers can be added to a single runner, allowing services to respond to both blockchain events and time-based triggers simultaneously.

Custom triggers implement the BackgroundKeeper trait for arbitrary trigger conditions. The trait defines a **start** method that spawns a background task and a **check_and_execute** method for individual iterations. The SDK provides several built-in keepers. The EpochKeeper monitors inflation epoch boundaries, triggering distribution claims when new epochs begin. The **StreamKeeper** monitors payment streams, triggering settlement when accumulated amounts reach configured thresholds. The RoundKeeper manages round-based protocols, triggering round advancement according to timing or completion conditions.

Developers implement custom keepers for their specific needs: monitoring external APIs, watching other blockchains, responding to market conditions, or any condition expressible in code.

The following example shows a custom keeper that monitors an external price feed:

```
use blueprint_sdk::tangle_evm::services::{BackgroundKeeper,
   KeeperConfig, KeeperHandle, KeeperResult};
use tokio::sync::broadcast;

struct PriceMonitorKeeper;

impl BackgroundKeeper for PriceMonitorKeeper {
    const NAME: &'static str = "price-monitor";

    fn start(config: KeeperConfig, mut shutdown: broadcast::
       Receiver<()>) -> KeeperHandle {
        let handle = tokio::spawn(async move {
            loop {
                tokio::select! {
                    _ = shutdown.recv() => break,
                    result = Self::check_and_execute(&config) => {
                        if let Err(e) = result { tracing::warn!("
                           Check failed: {e}"); }
                    }
                }
                tokio::time::sleep(config.round_check_interval).
                   await;
            }
            Ok(())
        });
        KeeperHandle { handle, name: Self::NAME }
    }

    async fn check_and_execute(config: &KeeperConfig) ->
       KeeperResult<bool> {
        // Monitor price and trigger actions when thresholds are
           crossed
        // Returns Ok(true) if action was taken, Ok(false) if no
           action needed
        Ok(false)
    }
}
```

The keeper spawns a background task that periodically checks conditions and responds to shutdown signals. The SDK manages the keeper's lifecycle through the KeeperHandle, coordinating startup, health monitoring, and graceful shutdown.

13.5 Error Handling and Reliability

Production operator software must handle failures gracefully. The SDK provides patterns for reliability.

Handler errors are caught and logged. A handler that returns `Err` does not crash the operator; the error is logged, and processing continues. For jobs requiring results, the operator may skip submission (accepting potential slashing) or submit an error indicator depending on blueprint design.

Submission failures trigger retries with exponential backoff. If result submission fails (network error, nonce conflict, insufficient gas), the consumer retries automatically. Configuration specifies retry limits and backoff parameters.

Chain reorganizations can invalidate submitted results. The SDK tracks submitted transactions and monitors for inclusion. If a reorg orphans a transaction, the consumer resubmits. This is particularly important for aggregated results where coordination already occurred.

Crash recovery uses persistent state for critical data. The SDK can checkpoint job processing progress, ensuring that a restarted operator does not miss jobs or double-process them. Configuration specifies checkpoint storage (file, database, or custom backend).

Graceful shutdown ensures clean termination. When receiving shutdown signals (SIGTERM, SIGINT), the SDK completes in-progress jobs, flushes pending submissions, and closes connections before exiting. This prevents data loss and ensures pending work is not abandoned.

13.6 Quality of Service

The Quality of Service (QoS) package provides comprehensive monitoring, metrics, and observability for operator services. Understanding this system helps developers build observable, reliable services.

13.6.1 QoS Architecture

The QoS system integrates multiple monitoring backends into a unified service:

Prometheus provides metrics collection and exposition. The SDK runs an embedded Prometheus server (configurable port) that exposes all metrics via the standard `/metrics` endpoint. Operators can scrape this endpoint with existing Prometheus infrastructure.

OpenTelemetry provides metrics export to OTLP-compatible backends (Grafana Cloud, Datadog, etc.). The SDK's OpenTelemetry exporter pushes metrics at configurable intervals, enabling integration with enterprise observability platforms.

Loki provides structured logging with labels. Log messages are tagged with service ID, blueprint ID, job ID, and custom labels, enabling powerful log querying and correlation with metrics.

Grafana provides dashboards and alerting. The SDK can provision Grafana dashboards automatically, showing job throughput, latency distributions, error rates, and system resource usage.

13.6.2 Built-in Metrics

The QoS package collects metrics automatically:

System metrics capture host-level resource usage: CPU utilization, memory consumption, disk I/O, and network traffic. These metrics help operators right-size infrastructure and identify resource constraints.

Blueprint metrics capture application-level behavior: jobs executed, execution times, success/failure rates, queue depths. The SDK automatically instruments job execution, recording duration and outcome for every job.

Blueprint status provides operational state: uptime, last heartbeat timestamp, status codes, and status messages. External systems query this status to determine service health.

13.6.3 Custom Metrics

Developers extend the built-in metrics with application-specific measurements.

The `MetricsProvider` trait defines the customization interface. Developers can implement this trait to provide entirely custom metrics backends. For most use cases, the built-in `EnhancedMetricsProvider` suffices with its extension points.

Custom key-value metrics are added via the `add_custom_metric()` method. These appear in both Prometheus and OpenTelemetry exports:

```
// Add custom metrics for business-specific measurements
qos_service.add_custom_metric("model_inference_count".into(),
    "1523".into()).await;
qos_service.add_custom_metric("cache_hit_rate".into(), "0.87".into
    ()).await;
```

Custom Prometheus collectors can be registered via the shared registry:

```
// Register a custom Prometheus gauge
let custom_gauge = prometheus::Gauge::new("my_custom_metric", "
    Description")?;
qos_service.shared_registry().register(Box::new(custom_gauge.clone
    ()))?;
custom_gauge.set(current_value);  // Update in job handlers
```

Custom OpenTelemetry instruments can be created via the exporter:

```
let latency_histogram = qos_service.opentelemetry_exporter()
    .meter().f64_histogram("custom_latency")
    .with_description("Custom operation latency").build();

latency_histogram.record(operation_time_ms, &[
    KeyValue::new("operation", "inference"),
    KeyValue::new("model", "llama-70b"),
]);
```

13.6.4 Job Instrumentation

The QoS package provides specific methods for recording job execution:

```
// Record successful job execution
qos_service.record_job_execution(job_id, execution_time, service_id
    , blueprint_id);

// Record job failure with error classification
qos_service.record_job_error(job_id, "timeout");
```

These methods update both Prometheus counters/histograms and OpenTelemetry metrics, providing consistent instrumentation across monitoring backends.

13.6.5 Heartbeats

Heartbeats are periodic liveness signals configured with interval (default 5 minutes), jitter to prevent thundering herd effects, and missed heartbeat thresholds. They prove operator liveness for slashing conditions, provide reputation data, and enable monitoring. For multi-operator services, heartbeats can be BLS-aggregated to reduce gas costs.

13.6.6 Configuration

QoS is configured via `QoSConfig`:

```
let qos_config = QoSConfig {
    heartbeat: Some(HeartbeatConfig::default()),
    metrics: Some(MetricsConfig {
        prometheus_server: Some(PrometheusServerConfig { port: 9090
            }),
        collection_interval_secs: 60,
        service_id: env.service_id,
        blueprint_id: env.blueprint_id,
        ..Default::default()
    }),
    loki: Some(LokiConfig { endpoint: "http://loki:3100".into(), ..
        Default::default() }),
    grafana: Some(GrafanaConfig { endpoint: "http://grafana:3000".
        into(), ..Default::default() }),
    manage_servers: false,  // Use external servers; true spawns
        Docker containers
    ..Default::default()
};
```

For local development, setting `manage_servers: true` causes the SDK to spawn Docker containers for Prometheus, Loki, and Grafana automatically, providing a complete observability stack without external infrastructure.

13.6.7 QoS-Based Slashing

The QoS system provides data; slashing provides consequences. Connecting these systems enables automatic enforcement of service level agreements. This section shows how developers tie QoS metrics to slashing conditions.

The Architecture: QoS metrics flow from operators to a monitoring service (on-chain or off-chain). When metrics violate thresholds, the monitoring service proposes a slash via the Tangle protocol. The service manager's `querySlashingOrigin` hook authorizes the monitoring service to propose slashes. After the dispute window, slashing executes automatically.

Example: Heartbeat-Based Slashing

The simplest QoS-slashing connection is heartbeat monitoring. Operators must submit heartbeats; missing too many triggers slashing.

```
contract HeartbeatSlashingManager is BlueprintServiceManagerBase {
    uint256 public constant MAX_MISSED_HEARTBEATS = 3;
    uint256 public constant HEARTBEAT_INTERVAL = 5 minutes;

    mapping(uint64 => mapping(address => uint256)) public
       lastHeartbeat;
    mapping(uint64 => mapping(address => uint256)) public
       missedCount;

    // Operators call this to record heartbeat
    function recordHeartbeat(uint64 serviceId) external {
        require(isOperatorInService(serviceId, msg.sender), "Not
           operator");
        lastHeartbeat[serviceId][msg.sender] = block.timestamp;
        missedCount[serviceId][msg.sender] = 0;
    }

    // Anyone can call to check and slash inactive operators
    function checkAndSlash(uint64 serviceId, address operator)
       external {
        uint256 lastBeat = lastHeartbeat[serviceId][operator];
        uint256 elapsed = block.timestamp - lastBeat;
        uint256 missedBeats = elapsed / HEARTBEAT_INTERVAL;

        if (missedBeats > MAX_MISSED_HEARTBEATS) {
            // Propose slash via Tangle protocol
            tangle.proposeSlash(serviceId, operator, 500); // 5%
               slash
        }
    }

    // This contract is authorized to propose slashes
    function querySlashingOrigin(uint64 serviceId) external view
       returns (address) {
        return address(this);
    }
}
```

Example: Latency-Based Slashing

More sophisticated QoS conditions require off-chain monitoring with on-chain verification. Here, a monitoring oracle tracks job latency and proposes slashes when SLAs are violated.

```
contract LatencySlashingManager is BlueprintServiceManagerBase {
    uint256 public constant MAX_LATENCY_MS = 5000;  // 5 second SLA
    uint256 public constant VIOLATION_THRESHOLD = 10; // 10
       violations = slash

    address public latencyOracle;  // Authorized to report latency
    mapping(uint64 => mapping(address => uint256)) public
       violations;
    // Oracle reports latency violations (called off-chain after
       monitoring)
    function reportLatencyViolation(
        uint64 serviceId,
        address operator,
        uint256 jobId,
        uint256 actualLatencyMs,
        bytes calldata proof  // Optional: ZK proof or signature
    ) external {
        require(msg.sender == latencyOracle, "Not oracle");
        require(actualLatencyMs > MAX_LATENCY_MS, "Not violation");

        violations[serviceId][operator]++;

        if (violations[serviceId][operator] >= VIOLATION_THRESHOLD)
            {
            tangle.proposeSlash(serviceId, operator, 1000); // 10%
               slash
            violations[serviceId][operator] = 0;  // Reset counter
        }
    }

    function querySlashingOrigin(uint64 serviceId) external view
       returns (address) {
        return address(this);  // This contract proposes slashes
    }
}
```

The operator-side SDK collects latency metrics via the QoS package. An off-chain monitoring service queries these metrics (via Prometheus endpoint or OpenTelemetry), compares against SLAs, and calls `reportLatencyViolation` when thresholds are exceeded.

Example: Availability-Based Slashing

For services requiring high availability, the service manager can track uptime and slash for excessive downtime.

```
contract AvailabilitySlashingManager is BlueprintServiceManagerBase
    {
    uint256 public constant MIN_UPTIME_PERCENT = 99;  // 99% SLA
    uint256 public constant MEASUREMENT_WINDOW = 1 days;

    struct UptimeRecord {
        uint256 windowStart;
        uint256 successfulChecks;
        uint256 totalChecks;
    }
    mapping(uint64 => mapping(address => UptimeRecord)) public
       uptime;

    // Called by monitoring system for each availability check
    function recordAvailabilityCheck(
        uint64 serviceId,
        address operator,
        bool wasAvailable
    ) external onlyMonitor {
        UptimeRecord storage record = uptime[serviceId][operator];
        // Reset window if expired
        if (block.timestamp > record.windowStart +
           MEASUREMENT_WINDOW) {
            // Check previous window before reset
            if (record.totalChecks > 0) {
                uint256 uptimePercent = (record.successfulChecks *
                   100) / record.totalChecks;
                if (uptimePercent < MIN_UPTIME_PERCENT) {
                    uint256 slashBps = (MIN_UPTIME_PERCENT -
                       uptimePercent) * 100;
                    tangle.proposeSlash(serviceId, operator, uint16
                       (slashBps));
                }
            }
            record.windowStart = block.timestamp;
            record.successfulChecks = 0;
            record.totalChecks = 0;
        }

        record.totalChecks++;
        if (wasAvailable) record.successfulChecks++;
    }
}
```

Connecting SDK Metrics to On-Chain Slashing

The full flow connecting SDK QoS to on-chain slashing:

```
// 1. Operator runs QoS-enabled service
let qos = QoSService::new(qos_config).await?;
qos.start_collection().await?;

// 2. Off-chain monitor queries operator metrics
// GET http://operator:9090/metrics
// Parse: job_latency_seconds, job_success_total, job_error_total

// 3. Monitor detects SLA violation
if avg_latency > SLA_THRESHOLD {
    // 4. Monitor calls service manager contract
    let call = service_manager.reportLatencyViolation(
        service_id,
        operator_address,
        job_id,
        measured_latency_ms,
        proof,
    );
    client.send_transaction(call).await?;
}
// 5. Service manager proposes slash via Tangle protocol
// 6. After dispute window, slash executes automatically
```

Design Considerations

When designing QoS-based slashing: calibrate thresholds to distinguish genuine misbehavior from normal variance; ensure measurement integrity through multiple independent monitors or cryptographic proofs; include evidence in slashing proposals for dispute resolution; make penalties proportional to harm; and consider grace periods for new operators.

13.7 BLS Aggregation Protocol

For jobs requiring multi-operator consensus, the SDK provides BLS signature aggregation. Understanding this protocol is essential for blueprints using aggregated results.

BLS (Boneh-Lynn-Shacham) signatures have a unique property: multiple signatures on the same message can be combined into a single signature that verifies as if all signers signed. This enables efficient on-chain verification: instead of verifying n ECDSA signatures (expensive), the contract verifies one aggregated BLS signature (single pairing check).

The aggregation protocol proceeds as follows:

1. **Job execution**: Each operator executes the job independently, producing a result.

2. **Result commitment**: Each operator signs their result using their BLS private key and broadcasts the signature to the P2P network. The message includes the result hash, service ID, job ID, and operator identifier.

3. **Signature collection**: The designated aggregator (typically the first operator in the service's operator list) collects signatures from the P2P network. The aggregator waits until sufficient signatures arrive (meeting the threshold) or a timeout expires.

4. **Aggregation**: The aggregator combines collected signatures into an aggregate signature. The aggregator also constructs a bitmap indicating which operators contributed signatures.

5. **Submission**: The aggregator submits the aggregated result to the contract. The contract verifies the aggregate signature against the operators' public keys and the result hash.

The aggregator role is not privileged in a security sense; an aggregator who submits an invalid aggregate signature fails verification. However, a non-responsive aggregator can delay result submission. Blueprints may designate backup aggregators or use round-robin aggregation to mitigate this.

Threshold configuration determines how many operators must sign for a valid aggregate. A 67% threshold requires signatures from operators representing 67% of the specified weight (either operator count or stake weight, depending on configuration). Higher thresholds provide stronger consensus but require more coordination.

BLS aggregation is configured at the blueprint level through the `IBlueprintServiceManager` interface. The service manager's `requiresAggregation` and `getAggregationThreshold` hooks determine whether jobs require aggregated signatures and what threshold is needed. The SDK coordinates signature collection and aggregation off-chain.

The following example shows a blueprint with multiple job types including aggregated results:

```
use blueprint_sdk::tangle_evm::TangleEvmLayer;

// Job IDs - aggregation is configured in the service manager
    contract
pub const INFERENCE_JOB: u8 = 0;      // Requires aggregation (set
    in contract)
pub const HEARTBEAT_JOB: u8 = 1;      // Single result

pub async fn inference(TangleEvmArg(req): TangleEvmArg<
   InferenceRequest>) -> TangleEvmResult<Completion> {
    let result = run_model(&req.prompt, req.max_tokens).await?;
    TangleEvmResult(Completion { text: result, model: MODEL_ID.into
        () })
}

pub async fn heartbeat(TangleEvmArg(_): TangleEvmArg<()>) ->
   TangleEvmResult<u64> {
    TangleEvmResult(std::time::SystemTime::now().duration_since(
        UNIX_EPOCH)?.as_secs())
}

pub fn router() -> Router {
    Router::new()
        .route(INFERENCE_JOB, inference.layer(TangleEvmLayer))
        .route(HEARTBEAT_JOB, heartbeat.layer(TangleEvmLayer))
}
```

Whether each job uses aggregation depends on what the service manager contract returns from `requiresAggregation(serviceId, jobIndex)`. For aggregated jobs, operators sign results with their BLS keys, signatures are collected off-chain, and the aggregated result is submitted via `submitAggregatedResult`. The threshold (e.g., 67% of stake) comes from the contract's `getAggregationThreshold` hook.

13.8 P2P Networking Layer

The SDK includes peer-to-peer networking built on libp2p. Peer discovery uses Kademlia DHT for global discovery and mDNS for local networks. Message propagation uses gossipsub for efficient, resilient pub/sub delivery. Direct connections enable point-to-point communication for coordination. Protocol extensibility allows blueprints to define custom protocols for checkpoint synchronization, state sharing, or domain-specific coordination.

13.9 Building Real Services: Use Case Examples

The SDK patterns described above combine into complete services. This section presents four representative use cases, each illustrating different SDK capabilities and architectural patterns.

13.9.1 Use Case 1: Price Oracle Service

A price oracle provides asset prices to DeFi protocols. The service must deliver fresh, accurate prices with high availability.

The oracle blueprint uses event triggers to watch for price requests and cron triggers for periodic price updates. Multiple operators independently fetch prices from approved sources and submit them. An aggregation strategy (median, weighted average, or custom) combines submissions into authoritative prices.

```
pub const FETCH_PRICE_JOB: u8 = 0;

pub async fn fetch_price(TangleEvmArg(pair): TangleEvmArg<AssetPair
   >) -> TangleEvmResult<PriceData> {
    // Fetch from multiple sources for redundancy
    let binance = fetch_binance(&pair).await.unwrap_or_default();
    let coinbase = fetch_coinbase(&pair).await.unwrap_or_default();
    let chainlink = fetch_chainlink(&pair).await.unwrap_or_default
       ();

    // Median aggregation for manipulation resistance
    let price = median(vec![binance, coinbase, chainlink]);

    TangleEvmResult(PriceData { pair, price, timestamp: now(),
       sources: 3 })
}
```

The service manager implements cross-operator verification: if any operator's submission deviates more than 0.5% from the median, it is flagged for review. Repeated deviations trigger slashing. This verification works because oracle outputs are objectively checkable against external sources.

13.9.2 Use Case 2: AI Inference Service

An AI inference service runs language models on operator infrastructure. Customers submit prompts; operators return completions. The service must verify that operators actually use the specified model rather than substituting cheaper alternatives.

The inference blueprint uses event triggers for incoming requests and background keepers for challenge-based verification. Operators run model servers locally or call external APIs depending on their configuration.

```
pub const INFERENCE_JOB: u8 = 0;

pub async fn inference(
    Context(config): Context<InferenceConfig>,
    TangleEvmArg(request): TangleEvmArg<InferenceRequest>,
) -> TangleEvmResult<Completion> {
    let model = ModelClient::new(&config.model_endpoint);
    let completion = model.complete(
        &request.prompt, request.max_tokens, request.temperature,
    ).await.map_err(|e| blueprint_sdk::Error::Other(e.to_string()))
       ?;

    TangleEvmResult(Completion {
        text: completion.text,
        usage: completion.tokens_used,
        model: config.model_id.clone(),
    })
}
```

Verification uses model fingerprinting challenges. The service manager maintains a database of (prompt, expected response pattern) pairs that distinguish the specified model from alternatives. Periodic challenge jobs test operators against this database. Operators who consistently fail challenges (producing responses inconsistent with the specified model) face slashing.

13.9.3 Use Case 3: Keeper Service

A keeper service monitors on-chain conditions and executes transactions when criteria are met. Examples include liquidation keepers for lending protocols, arbitrage keepers for DEX trading, and governance keepers for proposal execution.

The keeper blueprint uses custom BackgroundKeeper implementations that monitor specific conditions. When conditions are met, the keeper triggers job execution and result submission.

```
use blueprint_sdk::runner::BackgroundService;
use blueprint_sdk::runner::error::RunnerError;

pub struct LiquidationService {
    lending_protocol: Address,
    check_interval: Duration,
}

impl BackgroundService for LiquidationService {
    async fn start(&self) -> Result<Receiver<Result<(), RunnerError
       >>, RunnerError> {
        let (tx, rx) = oneshot::channel();
        let protocol = self.lending_protocol;
        let interval = self.check_interval;

        tokio::spawn(async move {
            loop {
                match fetch_undercollateralized(&protocol).await {
                    Ok(positions) => {
                        for position in positions {
                            // Submit liquidation transaction
                               directly
                            if let Err(e) = execute_liquidation(&
                               position).await {
                                tracing::warn!("Liquidation failed:
                                    {e}");
                            }
                        }
                    }
                    Err(e) => tracing::warn!("Failed to fetch
                       positions: {e}"),
                }
                tokio::time::sleep(interval).await;
            }
        });
        Ok(rx)
    }
}
```

Keeper services typically use direct result submission (no aggregation) because speed matters more than consensus. The first operator to detect a condition and submit a valid transaction earns the reward. Competition among operators ensures rapid response.

13.9.4 Use Case 4: Threshold Signature Service

A threshold signature service generates signatures without any single party holding the complete key. Applications include multi-party custody, cross-chain bridges, and institutional key management.

The threshold blueprint uses BLS aggregation with a twist: rather than aggregating results, operators participate in a distributed key generation (DKG) protocol and threshold signing ceremonies.

```
pub const SIGN_JOB: u8 = 0;

pub async fn sign(
    Context(state): Context<ThresholdState>,
    TangleEvmArg(message): TangleEvmArg<Bytes>,
) -> TangleEvmResult<PartialSignature> {
    let key_share = state.dkg_share.as_ref()
        .ok_or_else(|| blueprint_sdk::Error::Other("No DKG share".
           into()))?;
    let partial = key_share.partial_sign(&message)?;

    TangleEvmResult(PartialSignature {
        index: state.operator_index,
        signature: partial,
    })
}
```

The aggregator collects partial signatures from threshold participants and combines them into a complete signature. The threshold (e.g., 3-of-5) determines how many partial signatures are required. This service demonstrates how the SDK's aggregation infrastructure extends beyond simple voting to sophisticated cryptographic protocols.

13.9.5 Choosing Patterns for Your Service

Different services require different architectural patterns. The table below summarizes when to use each approach.

Service Type	Triggers	Submission
Oracle	Event + Cron	Aggregated (median)
AI Inference	Event	Direct or Aggregated
Keeper	BackgroundKeeper	Direct (speed-critical)
Threshold Crypto	Event	Aggregated (threshold)
Monitoring	Cron	Direct

Developers should consider: Does my service need consensus across operators? (Use aggregation.) Is speed critical? (Use direct submission.) Are outputs objectively verifiable? (Use cross-operator comparison.) Are outputs subjective? (Use challenge-response verification.)

13.10 Testing and Development

The SDK provides tooling for blueprint development and testing.

Local testing uses mock producers that generate test job calls without requiring blockchain infrastructure. Developers write unit tests that invoke handlers directly, verifying business logic without network dependencies.

Integration testing uses local blockchain instances (Anvil, Hardhat) with deployed contracts. Tests exercise the full flow from job submission through result verification. The SDK provides test utilities for contract deployment, job submission, and result verification.

Testnet deployment enables pre-production validation. Tangle maintains testnet infrastructure where operators can deploy blueprints, register services, and process real jobs with test tokens. Testnet behavior mirrors mainnet, providing realistic validation before production deployment.

Observability includes structured logging (with configurable verbosity), metrics export (Prometheus format), and distributed tracing (OpenTelemetry). Operators can integrate with their existing monitoring infrastructure to track blueprint performance and diagnose issues.

14 Verifiability and Detection

Slashing creates economic consequences for misbehavior, but consequences require detection. Without reliable detection, slashing becomes either toothless (violations go unpunished) or dangerous (false accusations destroy honest operators). This section presents the threat model, verification economics, concrete detection mechanisms, and the path toward building verifiable services.

14.1 Threat Model and Adversary Assumptions

Effective security design begins with explicit assumptions about adversaries. Tangle considers three adversary classes with distinct capabilities and motivations.

Rational adversaries seek to maximize economic return. They will cheat if and only if the expected value of cheating exceeds the expected value of honest behavior. For a rational adversary, the security condition is:

$$P(\text{detection}) \times \text{SlashAmount} > \text{GainFromCheating}$$

If this inequality holds, rational adversaries behave honestly. The protocol's primary defense against rational adversaries is ensuring that slashing amounts and detection probabilities make honesty the profit-maximizing strategy.

Byzantine adversaries may act irrationally, seeking to disrupt the system even at personal cost. They might be motivated by ideology, competitive sabotage, or external incentives invisible to the protocol. Defense against Byzantine adversaries requires bounding the damage any single actor can cause. Exposure limits bound individual operator losses. Dispute windows limit the speed of attacks. Governance provides recovery mechanisms for systemic failures.

Colluding adversaries coordinate across multiple protocol roles. An operator-customer collusion might generate fake jobs to inflate operator scores. An operator-developer collusion might design blueprints with weak verification to enable undetected cheating. Defense against collusion requires that verification not depend solely on parties with aligned incentives.

The protocol assumes that the majority of stake is held by honest or rational actors. It assumes that cryptographic primitives (hash functions, signatures, pairings) are secure. It assumes that the underlying blockchain provides eventual finality and censorship resistance. Within these assumptions, the protocol provides security guarantees; outside them, the guarantees may not hold.

14.2 The Economics of Verification

Verification is not free. Every detection mechanism consumes resources: compute for redundant execution, storage for audit logs, bandwidth for challenge-response protocols, human attention for dispute resolution. Effective verification design balances detection probability against verification cost.

Let C_v denote the cost of verification per service period, P_d the probability of detecting cheating given verification, S the slash amount, and G the gain from undetected cheating. The verification is economically justified when:

$$C_v < P_d \times S \times P_c$$

where P_c is the probability that an operator cheats absent verification. If operators rarely cheat, expensive verification may not be worthwhile. If cheating is profitable and common, verification becomes essential.

This analysis reveals several design principles. First, slash amounts should be calibrated to verification costs. High slashes enable profitable verification even with moderate detection probability. Second, verification can be probabilistic. Checking every execution is often unnecessary; random sampling can achieve sufficient deterrence at lower cost. Third, verification costs should scale with stakes. High-value services justify more expensive verification; low-value services may rely on reputation and sampling.

The protocol enables this flexibility through developer-defined slashing. Developers choose slash severities that make their verification economics work. A developer with cheap, high-probability detection can use lower slashes. A developer with expensive, lower-probability detection needs higher slashes to maintain deterrence.

14.3 Verification Approaches

Verification mechanisms fall into categories based on what they verify and how they achieve it.

14.3.1 Redundant Execution

The most straightforward verification approach runs the same computation on multiple operators and compares results. If operators produce different outputs for identical inputs, at least one is wrong.

Redundant execution requires careful design to prevent copying. If operators can see each other's results before submitting their own, a cheating operator simply copies honest results. The standard solution uses commit-reveal: operators first commit to result hashes, then reveal actual results after all commitments are collected. The BLS aggregation system supports this pattern through threshold signatures that require independent computation.

The number of redundant executors creates a tradeoff. More executors increase detection probability but also increase cost. For deterministic computations, two executors suffice to detect disagreement (though not to identify which is wrong). For computations with legitimate variance (such as AI model outputs with temperature sampling), more executors and tolerance thresholds may be needed.

14.3.2 Cryptographic Attestation

Some verification relies on cryptographic proofs rather than redundant execution. These proofs demonstrate that computation occurred correctly without re-executing it.

Zero-knowledge proofs enable a prover to convince a verifier that a computation was performed correctly without revealing the computation's inputs. ZK proofs are particularly valuable when inputs are sensitive (customer data) or when the computation is expensive to re-execute. The protocol's codebase includes integration points for SP1 ZK verification, enabling blueprints to require ZK proofs of correct execution.

Trusted Execution Environments (TEEs) provide hardware-based attestation. Intel SGX, AMD SEV, and ARM TrustZone create isolated execution environments that can produce cryptographic attestations of what code ran. TEE attestation proves that specific code executed in a protected environment, though it does not prove the code was correct, only that it was the expected code.

Remote attestation from model providers offers another path. If an AI model provider (such as OpenAI or Anthropic) offers cryptographic receipts proving that specific API calls occurred, these receipts can verify that operators used genuine model endpoints rather than cheaper substitutes.

14.3.3 Challenge-Response Protocols

Challenge-response verification tests operators with inputs whose correct outputs are known to the challenger but not the operator. Challengers maintain a database of (input, expected output) pairs and periodically submit these as regular jobs. Challenges must be indistinguishable from regular jobs, the database must be large enough to prevent memorization, and expected outputs must account for legitimate variance.

14.3.4 Reputation and Sampling

Reputation systems aggregate customer reports to identify consistently problematic operators. While single reports may be unreliable, patterns across many reports reveal signal.

Sampling-based verification randomly selects executions for intensive verification. If operators cannot predict which executions will be verified, they must assume any might be checked, enabling high detection rates at low average cost.

14.4 AI-Specific Verification Challenges

AI workloads present unique verification challenges that merit dedicated attention.

Model identity verification addresses whether operators run the models they claim. Unlike deterministic computations where identical inputs produce identical outputs, AI models exhibit variance. Two calls to the same model with identical inputs may produce different outputs due to sampling temperature, internal state, or version differences.

Fingerprinting approaches exploit the fact that different models have different output distributions. Given a set of probe inputs, the distribution of outputs forms a fingerprint. Comparing an operator's output distribution against known model fingerprints can identify which model is actually running. This detection is probabilistic: more probes yield higher confidence, but never certainty.

Performance verification addresses whether operators deliver advertised capabilities. Claims about tokens per second, latency, or context length can be tested empirically. The challenge is distinguishing genuine performance from temporarily inflated benchmarks. Operators might provision extra resources during known testing periods, then reduce capacity during normal operation.

Continuous random sampling addresses this: verify performance at unpredictable times, making sustained deception expensive. The heartbeat system provides infrastructure for this pattern: heartbeat responses can include timing proofs demonstrating response latency.

Isolation verification addresses whether sandboxes maintain their boundaries. This is perhaps the most challenging verification domain, as isolation failures may leave no externally visible trace. An agent that exfiltrates data to an attacker-controlled server leaves evidence only in network logs that the attacker controls.

Defense in depth addresses isolation verification: multiple overlapping controls (network policies, syscall filtering, process isolation) make breaches unlikely, while audit logging and anomaly detection make successful breaches more likely to produce detectable signals.

14.5 Connecting Verification to the Protocol

Verification mechanisms connect to protocol infrastructure through several integration points.

The `querySlashingOrigin` hook specifies which address may propose slashes. This might be the developer's verification contract, a decentralized oracle network, a dispute resolution system, or the service manager itself. By controlling the slashing origin, developers control who can trigger the economic consequences.

The dispute window provides time for investigation. When verification produces ambiguous results, the dispute period allows gathering additional evidence, consulting experts, or running additional tests. The accused operator can present counter-evidence. Protocol administrators can cancel slashes that prove fraudulent.

The slashing severity (basis points) calibrates economic consequences to violation severity. Minor violations (brief unavailability, slight performance degradation) might warrant small slashes. Major violations (data breach, model substitution, sustained cheating) might warrant large slashes or full stake forfeiture.

14.6 A Worked Example: Verifiable AI Sandbox

Consider building a verifiable AI sandbox service. The service promises customers that their agents run on specified models (say, Claude) with specified isolation guarantees.

The blueprint defines three job types: `execute` (run agent code), `heartbeat` (prove liveness), and `challenge` (respond to verification probes). The service manager implements verification through several mechanisms.

For model verification, the service manager maintains a challenge database of (prompt, expected response pattern) pairs known to produce distinctive outputs from Claude versus other models. Periodically, the manager submits challenge jobs through the same path as regular execution. Operators who consistently fail challenges face slashing.

For performance verification, heartbeat jobs include timing requirements. The heartbeat must return within a specified latency. Operators who miss heartbeats or exceed latency bounds accumulate strikes. Sufficient strikes trigger slashing.

For isolation verification, the service manager requires operators to run gVisor or Firecracker (declared at registration). The SDK's QoS package monitors for anomalies. Additionally, customer-reported isolation concerns trigger investigation during the dispute window.

The slashing configuration sets model substitution at 50% stake (severe, as it fundamentally violates the service promise), performance degradation at 5% stake (moderate, as it affects quality but not integrity), and isolation breach at 100% stake (maximum, as it potentially compromises customer security).

Quantified economics. Suppose a customer runs trading agents through this sandbox, with agents managing \$200,000 in assets. The profit from corruption (PfC) equals the maximum extractable value from compromised agents—approximately the assets under management, so PfC = \$200,000.

The service uses 3 operators, each staking \$100,000 with 80% exposure. The cost of corruption requires compromising all 3 operators (since any honest operator would detect model substitution via challenges), yielding CoC $= 3 \times \$100,000 \times 0.8 = \$240,000$.

Security ratio = \$240,000 / \$200,000 = 1.2. This exceeds 1 but falls below the recommended 1.5x threshold. The service should either reduce the assets customers can manage or increase operator stake requirements.

For detection probability, model fingerprinting challenges are submitted 10 times per day. Empirical testing suggests each challenge detects model substitution with 85% probability (due to some prompt overlap between models). Over a 7-day dispute window, detection probability is $1 - (1 - 0.85)^{70} \approx 99.99\%$, effectively ensuring detection. This high detection probability justifies the security ratio near 1.2—the effective expected cost of corruption is $0.9999 \times \$240,000 \approx \$240,000$.

This worked example illustrates how protocol primitives (hooks, jobs, slashing) compose with verification mechanisms (challenges, heartbeats, monitoring) to create a complete accountability system with quantified security guarantees.

14.7 A Second Example: Verifiable Oracle Service

Consider a price oracle service providing asset prices to DeFi protocols. The oracle promises freshness (prices updated within 5 minutes), accuracy (within 1% of true market price), and availability (99% uptime).

The verification approach differs from AI services because oracle outputs are checkable against external sources.

Source verification ensures operators fetch from approved data sources (major exchanges, data providers). The blueprint requires operators to provide signed data from at least 3 of 5 approved sources. Source signatures are submitted with each price update, enabling on-chain verification that data originated from legitimate sources.

Cross-operator comparison detects manipulation. Multiple operators submit prices independently. The service manager compares submissions; if one operator's price deviates by more than 0.5% from the median, that operator's submission is flagged. Repeated deviations trigger investigation and potential slashing.

External auditing provides additional verification. Anyone can compare oracle prices against public market data. If oracle prices systematically deviate from market prices, the community can submit slashing proposals with evidence.

The economics are quantified. Suppose the oracle serves DeFi protocols with \$100M in exposure. A manipulated price could enable \$10M in arbitrage profits (PfC = \$10M). The oracle requires 5 operators each staking \$3M with 100% exposure, yielding CoC = \$9M (corrupting 3-of-5). This security ratio of 0.9 is insufficient; the blueprint should either increase stake requirements or reduce exposure to protocols managing less capital.

Slashing amounts: source forgery (100%, as it completely compromises integrity), price deviation (10% per incident, accumulating with repeated violations), availability failure (1% per missed update, capped at 20%).

14.8 Formal Verification Properties

While full formal verification of verification mechanisms is beyond the scope of this paper, we state properties that verification systems should satisfy.

Property V1 (Detection Soundness): If an operator behaves honestly according to the protocol specification, the verification mechanism does not flag them for slashing with probability greater than ϵ (false positive rate).

Importance: False positives discourage honest operator participation. A verification mechanism with high false positive rate is unusable regardless of its detection power.

Property V2 (Detection Completeness): If an operator violates the protocol specification in manner M, the verification mechanism detects the violation with probability at least p_M (detection probability for violation type M).

Importance: Detection probability determines the effective cost of cheating. If p_M is low, the expected slashing cost $p_M \times S$ may not exceed the benefit V, making cheating rational.

Property V3 (Economic Sufficiency): For all violation types M with potential benefit V_M, the detection probability p_M and slash amount S_M satisfy $p_M \times S_M > V_M$.

Importance: This is the core security condition. Blueprints must configure detection and slashing to satisfy this inequality for all relevant violation types.

Property V4 (Verification Efficiency): The cost of verification C_v is bounded by a fraction α of the service value V_s: $C_v < \alpha \times V_s$.

Importance: Verification that costs more than the service provides is economically irrational. Practical systems require $\alpha < 0.1$ (verification costs less than 10% of service value).

Blueprints should analyze their verification mechanisms against these properties, quantifying false positive rates, detection probabilities, and verification costs. This analysis, documented in the blueprint specification, enables customers to make informed decisions about service security.

14.9 Standard Verification Libraries

The ecosystem benefits from shared verification infrastructure. Rather than each blueprint reimplementing common patterns, standard libraries can provide audited, tested verification components.

Oracle verification libraries might implement proof-of-data-source checking, freshness validation, and cross-source comparison. An oracle blueprint imports these libraries rather than building verification from scratch.

Compute verification libraries might implement commit-reveal schemes, redundant execution coordination, and result comparison with configurable tolerance. A compute blueprint uses these primitives to ensure correct execution.

Availability verification libraries might implement heartbeat scheduling, jitter management, and strike accumulation. Any blueprint requiring uptime guarantees can use these components.

AI verification libraries might implement model fingerprinting, challenge database management, and performance benchmarking. AI sandbox blueprints benefit from shared research into effective verification techniques.

These libraries reduce developer burden, improve security through concentrated review, and accelerate ecosystem development. The protocol team and community can collaborate on developing, auditing, and maintaining these shared resources.

Part IV

Products

The protocol provides infrastructure. Products provide value. This section describes the products built on Tangle, how they create value for users, and how that value flows through the protocol to all participants. Understanding this connection between products and protocol economics is essential to understanding how the network sustains and grows.

15 Value Accrual and the Product-Protocol Connection

Before describing individual products, it is worth understanding how value flows through the Tangle ecosystem. This section traces the path from user activity to participant rewards.

15.1 The Value Flow

Every interaction with Tangle products generates economic activity that accrues to network participants. The flow proceeds as follows.

A customer uses a product (the sandbox runtime, the workbench, or a third-party application built on Tangle). The product requests services from operators. The customer pays in tokens.

These payments split among developers, operators, and the protocol according to the configured distribution.

Operators provide the compute that makes products work. They stake tokens, run infrastructure, and earn fees from the services they provide. The more services they provide reliably, the more fees they earn. The more stake they accumulate (both self-stake and delegations), the more services they can participate in.

Developers create the blueprints that define how services work. They earn from both direct fee splits (when their blueprints are used) and from inflation rewards (proportional to their blueprint adoption). Successful developers whose blueprints see heavy usage accumulate significant rewards.

Delegators provide additional stake to operators they believe will perform well. They share in operator rewards proportional to their delegation. Delegators who pick reliable, high-performing operators earn more than those who pick poorly.

The protocol itself captures a governance-controlled fee (currently 10%) that funds ongoing development, security audits, and ecosystem growth. This fee is not rent extraction but reinvestment in the platform all participants share.

15.2 Network Effects and Growth Dynamics

Tangle exhibits positive network effects that compound over time.

More operators mean more available compute, which enables more products and more customer demand. More demand means more fees, which attracts more operators. This supply-side flywheel makes compute increasingly available and competitive.

More developers mean more blueprints, which enable more diverse products serving more use cases. More use cases attract more customers. More customers mean more fees for developers. This application-layer flywheel expands what is possible on the network.

More delegators mean more stake backing operators, which increases economic security. More security enables higher-value use cases. Higher-value use cases generate more fees, attracting more stake. This security flywheel makes the network progressively more trustworthy.

These flywheels interact. More security attracts enterprise customers. Enterprise customers demand specialized blueprints. Specialized blueprints require capable operators. Capable operators attract delegations. The ecosystem grows as a coherent whole rather than as isolated components.

15.3 Token Utility and Value Capture

The TNT token serves multiple functions that create demand and capture value.

Staking is required for operators to participate in the network. Operators must stake TNT to register, and their stake determines their capacity to serve customers. As demand for operator services grows, demand for stake grows, which creates demand for TNT.

Delegation allows passive token holders to earn yield by backing operators. Delegators stake TNT with operators and share in their rewards. The yield from delegation creates holding incentive independent of speculative appreciation.

Governance requires TNT for voting on protocol changes. Token holders control fee parameters, inflation schedules, upgrade decisions, and other protocol configuration. Governance power provides value to holders who care about protocol direction.

Payment for services can occur in TNT (at a discount) or other tokens. Services priced in TNT create direct utility demand. The payment discount incentivizes TNT usage over alternatives.

Fee distribution routes value to token holders through multiple channels: staking rewards, delegation rewards, and developer incentives all flow in TNT. The token is the medium through which the network distributes value to participants.

Together, these utilities create a coherent value proposition. TNT is not merely a speculative asset but a productive one that generates yield, confers governance rights, and captures value from network activity.

16 The Decentralized Sandbox Runtime

With the value flow established, we now describe the first major product: the decentralized sandbox runtime. This is where autonomous work actually executes, generating the activity that flows through the economic channels described above.

16.1 Motivation: Trust and Distribution

The question is not whether centralized cloud infrastructure works. It works well. The question is who controls it, who benefits from it, and whether alternatives should exist.

Centralized providers make decisions unilaterally. They set prices. They define terms of service. They choose which customers to serve. They capture the economic value their platforms generate. For many use cases, this is acceptable. For AI infrastructure that will underpin significant economic activity, concentration in few hands raises concerns.

Decentralized infrastructure offers an alternative. Independent operators compete to provide compute. Prices emerge from markets rather than corporate decisions. Economic value distributes to participants rather than concentrating in platform owners. No single entity can deny service or change terms unilaterally.

The sandbox runtime implements this alternative for AI agent execution. Operators run agents in isolated containers. Customers pay for execution with cryptographic accountability. The protocol coordinates without controlling.

16.2 Operator-Hosted Infrastructure

Operators host sandbox environments using industry-standard isolation technologies.

Docker provides container-based isolation with process, filesystem, and network separation. Containers are lightweight, well-understood, and widely deployed. Docker suits many workloads where the isolation guarantees are sufficient.

gVisor adds an additional isolation layer. gVisor intercepts system calls, providing a user-space kernel that limits the attack surface available to containerized workloads. This suits higher-security requirements where container escapes are a concern.

Firecracker and micro VMs provide hardware-level isolation. Micro VMs boot in milliseconds while providing the strong isolation of virtual machines. This suits workloads requiring the strongest isolation guarantees, where even kernel-level exploits should not compromise the host.

Operators choose isolation technologies based on their security requirements, performance needs, and the blueprints they serve. The protocol does not mandate specific technologies; it provides the economic framework within which operators make these decisions.

16.3 Isolation Guarantees

Regardless of isolation technology, certain guarantees must hold: process isolation preventing access to host resources or other sessions, filesystem isolation providing private storage with quotas, network isolation restricting external access according to policy, and resource limits bounding CPU, memory, and other consumption. Operators who fail to maintain these guarantees face slashing.

16.4 Sandbox as Sidecar

The sandbox serves as both primary execution environment and as a sidecar alongside other systems. As primary environment, it hosts AI agents performing tasks. As sidecar, it runs alongside DeFi protocols for AI-assisted monitoring, oracle networks for data processing, or keeper networks for decision logic. Any system needing secure, accountable AI execution can use operator-hosted sandboxes.

17 The Agentic Workbench

The agentic workbench is where autonomous work is authored and refined. It provides the creative environment for designing, testing, and deploying AI-powered workflows.

17.1 Vibe Coding Platform

Today, the workbench operates as a vibe coding platform for building projects against blockchain ecosystems. Users describe what they want to build; AI agents create the implementation. Agents work inside sandboxed containers, producing code that users review, refine, and deploy.

The platform handles the complexity of development environments. Pre-configured containers include SDKs, tools, and dependencies for target ecosystems. When a user starts a session targeting Ethereum development, the container already includes Foundry, Hardhat, OpenZeppelin libraries, and common testing tools. For Solana development, the container includes Anchor, the Solana CLI, and program scaffolding. Users focus on what they want to build, not on environment setup.

Sessions persist across interactions. Each session maintains state: the codebase, the conversation history, the agent's accumulated context about the project. Users return to ongoing projects, review agent work, provide feedback, and iterate. The development process becomes a collaboration between human intent and agent capability, where the agent remembers decisions made in previous sessions and builds on work already completed.

The technical architecture connects the workbench interface to the sandbox runtime through the protocol. When a user initiates a coding session, the workbench requests a service from an operator. The operator provisions a sandbox container and establishes a bidirectional connection. User inputs flow to the agent in the sandbox; agent outputs flow back to the workbench interface. The protocol handles payment, operator selection, and accountability. The user sees only a seamless coding environment; the decentralized infrastructure operates invisibly beneath.

17.2 Collaborative Features

The workbench supports multiplayer collaboration. Teams work on shared projects with synchronized state. Multiple people can observe agent progress, provide input, and coordinate on complex tasks. Unlike traditional IDEs where collaboration means editing the same files, workbench collaboration means directing the same agent infrastructure: multiple humans guiding multiple agents toward shared goals.

Real-time collaboration enables workflows that were previously impractical. One team member might define high-level architecture while another refines implementation details. A third might focus on testing and quality assurance. Agents work in parallel on different components, each supervised by the team member with relevant expertise. The workbench coordinates these parallel streams, handling merge conflicts, dependency management, and integration testing. The team assembles results into cohesive outputs without the coordination overhead that typically makes parallel software development difficult.

Collaboration extends to agent supervision. AI agents are capable but imperfect. They make mistakes, misunderstand requirements, and produce code that technically works but misses the point. Teams collectively review agent outputs, catch errors, and guide refinement. One team member might notice a security issue another missed. Another might recognize a design pattern violation. Shared oversight improves quality beyond what any individual might achieve, combining human judgment with agent productivity.

The collaboration model also enables knowledge transfer. Junior developers work alongside seniors, observing how experienced engineers direct agents, what prompts produce good results, and how to recognize and correct common errors. The workbench becomes not just a productivity tool but a training environment where skills propagate through observation and practice.

17.3 Vibe Working: Beyond Code

The workbench is expanding beyond code to general knowledge work. Research, analysis, writing, and other cognitive tasks are natural extensions of the vibe coding paradigm. The same infrastructure that enables sandboxed code execution enables sandboxed research, analysis, and content creation.

Research agents gather, synthesize, and present information. A user investigating a market opportunity might deploy a research agent to analyze competitors, summarize market reports, identify regulatory considerations, and present findings in structured format. The agent runs in a sandbox with web access permissions, maintaining session state across multiple queries and building context over time. The same economic and accountability mechanisms that govern code execution govern research execution: operators stake assets, customers pay fees, and slashing conditions apply.

Analysis agents process data and generate insights. A user with a dataset might deploy an analysis agent to perform statistical analysis, identify patterns, generate visualizations, and suggest interpretations. The agent operates in a sandbox with access to the uploaded data, using analysis libraries (pandas, numpy, matplotlib) to produce results. Complex analyses that would take hours manually complete in minutes with agent assistance.

Writing agents produce reports, documentation, and communications. A user who has completed analysis might deploy a writing agent to transform findings into polished output: executive summaries, technical documentation, or presentation slides. The agent works from the analysis outputs, maintaining consistency and style across documents.

The vision is a unified environment for all AI-assisted work. Users engage with agents across task types without context switching between different tools and platforms. A single session might involve research (gathering information), analysis (processing data), coding (building tools), and writing (presenting results). The workbench becomes the operating surface for human-AI collaboration, flexible enough to accommodate whatever task mix the user requires.

17.4 The Parallel Agents Paradigm

Traditional AI interfaces present single-threaded conversation: one prompt, one response, then another prompt. This sequential model reflects the history of chat interfaces but does not reflect how complex work actually happens. Real projects involve exploration, comparison, and parallel investigation. The workbench introduces parallel agents: multiple agents working simultaneously on related tasks.

Users can spawn agents to handle subtasks. A primary agent working on a smart contract might spawn sub-agents to research gas optimization patterns, investigate similar implementations, and draft test cases. Each sub-agent runs independently, with its own sandbox and context, reporting results back to the primary agent or directly to the user.

Users can fork agents to explore alternative approaches. Facing an architectural decision (should this system use an upgradeable proxy or be immutable?), the user forks the current context and directs each fork down a different path. Both approaches develop in parallel. The user observes progress, compares results, and picks the winner. The losing fork is discarded; the winning fork continues.

The workbench presents parallel activity on a spatial canvas showing real-time progress. Users see all active agents, their status, their recent outputs, and their relationships to each other. This visual representation makes parallel work manageable rather than overwhelming. Users pick winners from parallel explorations, merge successful results into unified outputs, and maintain coherence across multiple workstreams.

This paradigm suits complex work where exploration benefits from parallelism. Software architecture involves tradeoffs that are difficult to evaluate abstractly. By actually implementing multiple approaches and comparing results, users make better decisions. Rather than committing early to an approach that might prove wrong, users explore the decision space and commit only when evidence supports the choice.

17.5 The Evaluation Loop

Every execution generates traces: structured logs of what agents did, what inputs they received, what outputs they produced, and how long operations took. These traces feed evaluation systems that improve future performance, creating a continuous improvement cycle that compounds over time.

Traces capture multiple dimensions: inputs (prompts, context, parameters), outputs (code, analysis, content), behavior (tool calls, API requests, reasoning chains), timing (latency at each stage), and outcomes (accepted, rejected, or modified).

Evaluation correlates traces with success metrics to answer questions with data: which prompt structures produce better results, which model configurations work for which tasks, which operators deliver lower latency. Insights inform prompt optimization, model selection, and workflow refinement.

This creates a flywheel: more usage generates more traces, which improve the system, which attracts more usage. Early users benefit from the infrastructure; later users benefit from the accumulated learning. The evaluation loop transforms raw execution into accumulated capability, making the workbench progressively more effective at every task type it supports.

18 The Ecosystem in Action

The sandbox runtime and workbench are first-party products, but they represent just the beginning of what the Tangle ecosystem enables. This section describes how these products interact, how third-party developers extend them, and how the entire ecosystem creates compounding value.

18.1 Product Interactions

The sandbox and workbench work together but also function independently. Users can design workflows in the workbench and deploy to sandbox runtime, or interact with sandbox directly through APIs. Enterprise systems, automated pipelines, and third-party applications access sandbox compute through standard interfaces. The workbench itself consumes sandbox compute, making it simultaneously a product and a protocol customer.

18.2 Third-Party Extensions

The protocol's openness enables third-party products: specialized workbenches for specific domains (legal, medical, financial), infrastructure tools for operators (monitoring, scaling, fleet management), and aggregator services helping customers find operators. Each third-party product creates and captures value while the protocol captures its fee regardless of which products generate activity.

18.3 Use Cases Enabled

The combination of protocol, products, and ecosystem extensions enables use cases that would be difficult or impossible with centralized alternatives.

Confidential AI for enterprises. A pharmaceutical company needs AI analysis of proprietary clinical data. They cannot send this data to cloud providers whose security practices they do not control. With Tangle, they select operators with verified security credentials, require specific isolation technologies, and maintain cryptographic evidence of proper handling. The economic penalties for data breach provide accountability beyond legal contracts.

Verifiable AI for high-stakes decisions. A trading firm needs AI-assisted analysis for investment decisions. They require confidence that the analysis actually used the specified model and data. With Tangle, they deploy services with verification mechanisms that detect model substitution or data manipulation. The execution trail provides audit evidence for regulators and investors.

Collaborative AI for distributed teams. A research consortium spans multiple institutions with different governance requirements. Each institution needs control over its data and compute. With Tangle, each institution can run operator infrastructure within its boundaries while participating in collaborative workflows coordinated by the protocol. No single institution controls the shared infrastructure.

Autonomous agents with accountability. A user wants to deploy AI agents that take real-world actions: making purchases, sending communications, executing transactions. These agents need to operate autonomously but with clear accountability. With Tangle, agents run in sandboxed environments with logged actions and economic guarantees. Misbehaving agents trigger consequences for the operators running them.

Developer monetization at scale. A developer creates a useful blueprint and wants to earn from its adoption. With Tangle, they deploy the blueprint once and earn from every service instantiation across all operators. The inflation mechanism provides additional rewards proportional to adoption. The developer captures value from their creation without building and maintaining infrastructure.

These use cases illustrate the breadth of what the ecosystem enables. Each use case generates activity that flows through the protocol's economic channels, creating value for all participants.

18.4 Measuring Ecosystem Health

Governance monitors key metrics to maintain ecosystem health: operator diversity (geographic and organizational), blueprint diversity (use case coverage), customer growth and retention, fee volume, and stake growth. These metrics guide parameter adjustments (inflation rates, fee structures, minimum stakes) to ensure sustainable growth benefiting all participants.

Part V

Conclusion

19 Roadmap

Development proceeds through phases targeting progressive capability and decentralization. Rather than presenting speculative timelines, this section describes the current state, planned developments, and the dependencies between them.

19.1 Current State

The protocol has achieved several foundational milestones.

The **core protocol contracts** are deployed and audited. The Tangle contract with its mixin architecture, the MultiAssetDelegation contract for staking and delegation, and the governance contracts are operational. These contracts implement the full lifecycle described in this paper: blueprint registration, service creation, job execution, payment distribution, and slashing.

The **Blueprint SDK** is complete with all major components: event-based and cron-based triggers, job handlers with extractors, direct and aggregated consumers, P2P networking, and the quality of service package. Developers can build and deploy services using documented patterns and production-ready tooling.

The **workbench** operates as a vibe coding platform supporting blockchain development. Users can create projects, direct AI agents, and produce deployable code. Session persistence, environment management, and basic collaboration features are functional.

The **operator network** includes initial operators running sandbox infrastructure. These operators demonstrate the economic model while the network expands to production scale.

19.2 Near-Term Development

Near-term priorities expand capability and adoption.

Multiplayer collaboration will enable real-time shared workbench sessions. Multiple users will observe and direct agents simultaneously, with synchronized state and coordinated outputs. Technical implementation involves operational transform or CRDT-based state synchronization, presence indicators, and permission management.

Vibe working expansion will extend the workbench beyond code to research, analysis, and writing tasks. This requires new agent configurations, expanded tool integration (web search, document processing, data analysis), and UI adaptations for non-code outputs.

Operator network growth will expand geographic and organizational diversity. Incentive programs will attract operators across regions. Improved tooling will reduce operational burden. Documentation and support will lower entry barriers.

Additional isolation technologies will integrate gVisor and Firecracker alongside Docker. This requires SDK updates for isolation configuration, operator tooling for managing multiple isolation backends, and verification mechanisms for isolation compliance.

Standard verification libraries will provide audited implementations of common patterns: oracle verification, compute verification, availability monitoring. These libraries reduce developer burden and improve ecosystem security.

19.3 Longer-Term Development

Longer-term development targets maturity and decentralization.

Full governance activation will transfer protocol control to token holders. Initial governance parameters (conservative thresholds, guardian oversight) will relax as the community demonstrates governance capability. The guardian multisig will progressively cede powers until governance operates fully autonomously.

Cross-chain deployment will extend the protocol to additional chains. L2s (Arbitrum, Base, Optimism) provide lower transaction costs for high-frequency operations. Alternative L1s may follow if demand justifies.

Ecosystem grants will fund developer tooling, verification research, and operator infrastructure. Treasury allocation to grants will follow governance approval, with transparent processes for application and evaluation.

Advanced verification research will explore ZK proofs for compute verification, TEE integration for confidential execution, and formal verification of critical protocol properties. Research outcomes will flow into standard libraries.

Protocol upgrades will add capabilities based on operational experience. Likely candidates include more sophisticated pricing mechanisms, additional delegation modes, enhanced slashing parameters, and integration with evolving restaking standards.

19.4 Dependencies and Sequencing

Multiplayer collaboration requires stable single-user experience. Vibe working expansion requires proven agent reliability. Governance activation requires sufficient token distribution. Cross-chain deployment requires mainnet stability. Near-term items are largely independent; longer-term items build on earlier foundations.

19.5 Adaptability

The roadmap is not fixed. Governance adjusts priorities as conditions change. Success is measured by concrete metrics: operator count, service volume, developer adoption, and stake growth.

20 Conclusion

This paper has presented Tangle Network, the operating layer for autonomous work. As AI transitions from assistant to workforce, infrastructure questions become paramount. Who operates the systems? Who captures the value? Who decides the rules?

Tangle answers these questions with decentralization. The protocol enables developers to define services, operators to provide compute, delegators to contribute stake, and customers to access capabilities. Economic incentives align behavior without centralized control. Cryptographic mechanisms ensure accountability without trusted intermediaries.

The technical contributions include the blueprint system enabling arbitrary service types without protocol modification, the request-for-quote mechanism creating markets for service pricing, O(1) algorithms for staking, slashing, and rewards that scale without iteration, the hook system enabling expressive crypto-economic customization, and the developer platform for building verifiable distributed services.

The products demonstrate what becomes possible. The sandbox runtime provides secure AI execution on decentralized infrastructure. The workbench provides a collaborative environment for authoring autonomous work. Together they show that decentralized AI infrastructure is not merely theoretical but operational.

The comparison to existing restaking protocols clarifies Tangle's position in the broader ecosystem: focused on compute services where AI infrastructure demands will grow, emphasizing developer expressiveness where arbitrary service types must be possible, and providing integrated products where working demonstrations matter.

The future belongs to those who build it. The patterns established today (who operates infrastructure, who captures value, who sets the rules) will shape the autonomous economy for decades. Tangle provides infrastructure for building an AI future that is owned by many rather than few, governed by participants rather than corporations, and accessible to anyone willing to contribute.

This is the operating layer for autonomous work.

References

1. Boneh, D., Lynn, B., & Shacham, H. (2001). Short signatures from the Weil pairing. *ASIACRYPT 2001.*

2. Buterin, V. (2014). A next-generation smart contract and decentralized application platform. *Ethereum Whitepaper.*

3. Buterin, V. et al. (2020). Combining GHOST and Casper. *arXiv preprint arXiv:2003.03052.*

4. EigenLayer. (2023). EigenLayer: The Restaking Collective. *EigenLayer Whitepaper.*

5. Symbiotic. (2024). Symbiotic: A Permissionless Shared Security System. *Symbiotic Documentation.*

6. EIP-712: Typed structured data hashing and signing. *Ethereum Improvement Proposals.*

7. ERC-4626: Tokenized Vaults. *Ethereum Improvement Proposals.*

8. ERC-7540: Asynchronous Tokenized Vaults. *Ethereum Improvement Proposals.*

9. Firecracker: Secure and fast microVMs for serverless computing. *Amazon Web Services.*

10. gVisor: Application Kernel for Containers. *Google Open Source.*

11. OpenZeppelin. (2024). OpenZeppelin Contracts. https://github.com/OpenZeppelin/openzeppelin-contracts

12. Russell, S., & Norvig, P. (2020). Artificial Intelligence: A Modern Approach (4th ed.). Pearson.

13. Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Yellow Paper.*

14. Nakamoto, S. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System. *Bitcoin Whitepaper.*

15. Szabo, N. (1997). Formalizing and Securing Relationships on Public Networks. *First Monday.*

16. Roughgarden, T. (2021). Transaction Fee Mechanism Design. *EC '21: Proceedings of the 22nd ACM Conference on Economics and Computation.*

17. Daian, P., et al. (2020). Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. *2020 IEEE Symposium on Security and Privacy.*

18. Gudgeon, L., et al. (2020). DeFi Protocols for Loanable Funds: Interest Rates, Liquidity and Market Efficiency. *AFT '20.*

19. libp2p. Modular peer-to-peer networking stack. https://libp2p.io/

20. Tokio. An asynchronous runtime for Rust. https://tokio.rs/

21. Yao, A. C. (1982). Protocols for secure computations. *FOCS '82.*

22. Ben-Sasson, E., et al. (2014). Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. *USENIX Security '14.*

23. Costan, V., & Devadas, S. (2016). Intel SGX Explained. *IACR Cryptology ePrint Archive.*

24. Castro, M., & Liskov, B. (1999). Practical Byzantine Fault Tolerance. *OSDI '99.*

25. Gennaro, R., Jarecki, S., Krawczyk, H., & Rabin, T. (1999). Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. *EUROCRYPT '99.*

26. Boldyreva, A. (2003). Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. *PKC '03.*

27. Brown, T., et al. (2020). Language Models are Few-Shot Learners. *NeurIPS 2020.*

28. Anthropic. (2024). Claude: Constitutional AI and Helpful, Harmless, Honest Assistants. *Anthropic Research.*

29. OpenAI. (2023). GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774.*