Tangle Blueprint SDK

Developer Guide

Building Decentralized Services with Rust

Tangle Foundation

January 2025

Abstract

The Tangle Blueprint SDK provides the tooling for building decentralized services that interact with the Tangle protocol. This guide covers the SDK's architecture, design patterns, and the path from concept to deployed service. Written for developers familiar with Rust and blockchain concepts, it presents practical examples ranging from simple handlers to production-grade services including oracles, AI inference, keepers, and threshold cryptography.

Contents

1 Introduction

Tangle enables developers to define computational services that run across a network of independent operators. The Blueprint SDK is the primary toolkit for building these services in Rust.

This guide assumes familiarity with:

- Rust programming and async/await patterns
- Basic blockchain concepts (transactions, events, contracts)
- Cryptographic primitives (signatures, hashes)

1.1 What You'll Build

A blueprint defines a service type. When deployed, operators register to provide that service, and customers create service instances by selecting operators and configuring parameters. Your blueprint code runs on operator infrastructure, processing jobs and submitting results.

1.2 Why Rust

The SDK is written in Rust for several reasons:

- **Performance**: Critical for operators processing high-throughput workloads. Rust compiles to native code with zero-cost abstractions.
- **Safety**: Prevents memory errors and race conditions. For operator software handling valuable stake and sensitive data, safety is non-negotiable.
- **Async runtime**: Tokio provides efficient concurrent execution for handling blockchain events, P2P networking, and customer requests simultaneously.
- **WebAssembly compilation**: Enables portable execution for sandboxed environments.
- **Ecosystem compatibility**: Excellent integration with blockchain tooling (alloy, libp2p, ark-bn254).

2 Architecture Overview

The SDK is organized into composable components.

2.1 Core Components

Producers generate job calls from external events:

- `TangleProducer` watches blockchain events
- `CronJob` generates calls on schedules
- Custom producers watch APIs, queues, or any event source

Router dispatches job calls to handlers. When a job arrives, the router examines the job index and invokes the corresponding function.

Handlers implement job-specific logic. Each handler receives context (extractors) and returns a result. Handlers are async functions.

Consumer submits results to the blockchain. For aggregated results, the consumer coordinates with other operators to collect signatures.

Background services run alongside job processing:

- `BackgroundService` for general long-running tasks
- `BackgroundKeeper` for Tangle lifecycle automation

P2P layer provides inter-operator communication for quote dissemination, signature aggregation, and custom coordination.

2.2 Execution Flow

1. Producer detects event (blockchain, cron, custom)
2. Producer generates job call
3. Router dispatches to appropriate handler
4. Handler executes and returns result
5. Consumer submits result (coordinating aggregation if needed)

Throughout this flow, background services continue running and the P2P layer handles coordination.

3 Your First Blueprint

Let's build a minimal blueprint that squares numbers.

3.1 Project Setup

Create a new Rust project:

```
cargo new my-blueprint
cd my-blueprint
```

Add dependencies to `Cargo.toml`:

```
[dependencies]
blueprint-sdk = { version = "0.8", features = ["tangle-evm"] }
tokio = { version = "1", features = ["full"] }
```

3.2 The Complete Example

```
use blueprint_sdk::tangle::extract::{TangleArg, TangleResult};
use blueprint_sdk::tangle::{TangleConsumer, TangleProducer, TangleLayer
    };
use blueprint_sdk::runner::{BlueprintRunner, config::
    BlueprintEnvironment};
use blueprint_sdk::runner::tangle::TangleConfig;
use blueprint_sdk::Router;

pub const SQUARE_JOB: u8 = 0;

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

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

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

    BlueprintRunner::builder(TangleConfig::default(), env)
        .router(router())
        .producer(TangleProducer::new(service_id))
        .consumer(TangleConsumer::new())
        .run()
        .await
}
```

3.3 Understanding the Code

The SDK uses an **extractor pattern** inspired by web frameworks:

- `TangleArg<T>` extracts ABI-encoded inputs from job calls
- `TangleResult<T>` wraps outputs for ABI-encoded submission
- `TangleLayer` handles encoding/decoding transformation

Job functions are plain async functions. The main function composes SDK components: producer watches for jobs, router directs to handlers, consumer submits results.

4 The Hook System

Blueprints customize protocol behavior through hooks, functions called at specific lifecycle points.

4.1 Lifecycle Hooks

onRegister validates operator registrations. A blueprint for AI services might require operators to prove GPU capability.

onRequest validates service requests. A membership service might verify the customer's eligibility.

onApprove processes operator approvals. A DeFi service might lock collateral or initialize state.

onSlash executes custom slashing logic. A threshold service might redistribute key shares.

onServiceTermination handles cleanup. A subscription service might finalize billing.

Hooks are implemented in Solidity by extending `BlueprintServiceManagerBase`:

```
contract MyBlueprintBSM is BlueprintServiceManagerBase {
    // Track registered operators
    mapping(address => bool) public isRegistered;
    mapping(address => bytes) public operatorMetadata;

    // Validate operator registration
    function onRegister(
        address operator,
        bytes calldata inputs
    ) external payable override onlyFromTangle {
        // Decode and validate registration inputs
        (uint256 gpuMemoryGB, string memory region) =
            abi.decode(inputs, (uint256, string));

        require(gpuMemoryGB >= 24, "Minimum 24GB GPU required");
        require(bytes(region).length > 0, "Region required");

        isRegistered[operator] = true;
        operatorMetadata[operator] = inputs;

        emit OperatorRegistered(operator, gpuMemoryGB, region);
    }

    // Process service requests
    function onRequest(
        uint64 requestId,
        address requester,
        address[] calldata operators,
        bytes calldata requestInputs,
        uint64 ttl,
        address paymentAsset,
        uint256 paymentAmount
    ) external payable override onlyFromTangle {
        // Validate the request parameters
        require(operators.length >= 3, "Minimum 3 operators");
        require(ttl >= 1 days, "Minimum 1 day TTL");

        // Store request for later reference
        requests[requestId] = RequestData({
            requester: requester,
            operatorCount: operators.length,
            ttl: ttl
        });
    }

    // Handle slashing events
    function onSlash(
        uint64 serviceId,
        bytes calldata offender,
        uint8 slashPercent
    ) external override onlyFromTangle {
        // Custom logic: redistribute key shares if threshold service
        address operator = abi.decode(offender, (address));
        emit OperatorSlashed(serviceId, operator, slashPercent);

        // Trigger key resharing protocol
        if (slashPercent >= 50) {
            initiateKeyReshare(serviceId, operator);
        }
    }
}
```

4.2 Slashing Customization

Blueprints define their own slashing conditions:

`querySlashingOrigin` specifies who may propose slashes (governance, specific contract, anyone with evidence).

`queryDisputeOrigin` specifies who may dispute (the accused operator, governance, delegated arbitrators).

Custom verification contracts implement evidence validation, dispute resolution, and penalty calculation.

4.3 Membership Customization

For dynamic membership services:

queryMinOperators/queryMaxOperators bound the operator set size.

queryExitDelay specifies how long operators must wait before leaving.

Custom logic can implement reputation-based entry, stake-weighted selection, or geographic distribution requirements.

4.4 Job Configuration

requiresAggregation determines whether results need multi-operator consensus.

getAggregationThreshold specifies the required agreement percentage (by count or stake weight).

5 Triggers in Depth

Triggers initiate job execution. The SDK provides multiple trigger types.

5.1 Event Triggers

`TangleProducer` watches blockchain events via WebSocket:

- Subscribes to Tangle contract events
- Filters for configured service
- Extracts job parameters
- Handles reconnection and reorgs

5.2 Cron Triggers

`CronJob` executes on schedules using standard cron expressions with seconds precision:

```
// Every second
"* * * * * *"

// Every 5 minutes
"0 */5 * * * *"

// Daily at midnight
"0 0 0 * * *"
```

Cron triggers suit periodic tasks: heartbeats, maintenance, report generation, API polling.

5.3 Custom Triggers (BackgroundKeeper)

For lifecycle automation, implement `BackgroundKeeper`. The SDK provides built-in keepers:

- `EpochKeeper` for inflation distribution
- `StreamKeeper` for payment stream settlement
- `RoundKeeper` for round-based protocols

```rust
use blueprint_sdk::tangle::services::{BackgroundKeeper, KeeperConfig,
    KeeperHandle};
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)
        -> blueprint_sdk::tangle::services::KeeperResult<bool>
    {
        // Monitor conditions and trigger actions
        Ok(false)
    }
}
```

6 BLS Aggregation

Multi-operator services use BLS signature aggregation for efficient consensus.

6.1 When to Use Aggregation

Use aggregation when:

- Multiple operators must agree on a result
- You need Byzantine fault tolerance
- Verification cost matters (BLS aggregates cheaply)

Skip aggregation when:

- Speed is paramount (first-to-submit wins)
- Single-operator services
- Outputs are independently verifiable

6.2 Aggregation Flow

1. Each operator computes result and signs with BLS key
2. Operators broadcast signatures via P2P
3. Aggregator collects signatures until threshold met
4. Aggregated signature submitted with combined result
5. Contract verifies aggregate signature on-chain

6.3 Configuration

The service manager contract controls aggregation:

- `requiresAggregation(serviceId, jobIndex)` returns whether job needs aggregation
- `getAggregationThreshold(serviceId)` returns required percentage

7 Use Case Examples

7.1 Price Oracle Service

A price oracle provides asset prices to DeFi protocols.

```rust
use blueprint_sdk::tangle::extract::{TangleArg, TangleResult};
use std::time::{SystemTime, UNIX_EPOCH};

pub const FETCH_PRICE_JOB: u8 = 0;

pub async fn fetch_price(
    TangleArg(pair): TangleArg<AssetPair>
) -> TangleResult<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]);
    let timestamp = SystemTime::now()
        .duration_since(UNIX_EPOCH).unwrap().as_secs();

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

Triggers: Event (price requests) + Cron (periodic updates)
Submission: Aggregated (median across operators)
Verification: Cross-operator comparison; deviations trigger review

7.2 AI Inference Service

An AI inference service runs language models for customers.

```rust
use blueprint_sdk::tangle::extract::{TangleArg, TangleResult};
use blueprint_sdk::extract::Context;

pub const INFERENCE_JOB: u8 = 0;

pub async fn inference(
    Context(config): Context<InferenceConfig>,
    TangleArg(request): TangleArg<InferenceRequest>,
) -> TangleResult<Completion> {
    let model = ModelClient::new(&config.model_endpoint);
    let completion = model.complete(
        &request.prompt,
        request.max_tokens,
        request.temperature,
    ).await?;

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

Triggers: Event (inference requests)
Submission: Direct or aggregated depending on requirements
Verification: Model fingerprinting challenges detect substitution

7.3 Keeper Service

A keeper monitors on-chain conditions and executes transactions.

```
use blueprint_sdk::runner::{BackgroundService, error::RunnerError};
use alloy_primitives::Address;
use std::time::Duration;
use tokio::sync::oneshot::Receiver;

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 {
                            if let Err(e) = execute_liquidation(&
                               position).await {
                                tracing::warn!("Liquidation failed: {e}
                                   ");
                            }
                        }
                    }
                    Err(e) => tracing::warn!("Failed to fetch: {e}"),
                }
                tokio::time::sleep(interval).await;
            }
        });
        Ok(rx)
    }
}
```

Triggers: BackgroundKeeper (condition monitoring)
Submission: Direct (speed-critical)
Verification: Transaction success is self-evident

7.4 Threshold Signature Service

A threshold service generates signatures without any party holding the complete key.

```
use blueprint_sdk::tangle::extract::{TangleArg, TangleResult};
use blueprint_sdk::extract::Context;

pub const SIGN_JOB: u8 = 0;

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

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

Triggers: Event (signing requests)
Submission: Aggregated (threshold combination)
Verification: Signature validity is cryptographically verifiable

7.5 Pattern Selection Guide

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

8 Error Handling and Reliability

Production software must handle failures gracefully. The SDK provides structured error types and recovery patterns.

8.1 Handler Errors

Handler errors are caught and logged. A handler returning `Err` does not crash the operator. Depending on blueprint design, the operator may skip submission or submit an error indicator.

```
use blueprint_sdk::tangle::extract::{TangleArg, TangleResult};
use thiserror::Error;
```

```rust
#[derive(Error, Debug)]
pub enum JobError {
    #[error("External API unavailable: {0}")]
    ApiUnavailable(String),
    #[error("Invalid input: {0}")]
    InvalidInput(String),
    #[error("Computation timeout")]
    Timeout,
}

pub async fn fetch_price(
    TangleArg(pair): TangleArg<AssetPair>
) -> Result<TangleResult<PriceData>, JobError> {
    // Validate input
    if pair.base.is_empty() || pair.quote.is_empty() {
        return Err(JobError::InvalidInput("Empty asset pair".into()));
    }

    // Fetch with timeout
    let price = tokio::time::timeout(
        Duration::from_secs(10),
        fetch_from_sources(&pair)
    ).await.map_err(|_| JobError::Timeout)??;

    Ok(TangleResult(price))
}
```

8.2 Submission Failures

The consumer retries with exponential backoff on network errors, nonce conflicts, or insufficient gas. Configuration specifies retry limits and backoff parameters.

```rust
// Consumer automatically handles retries
// Configure via TangleConsumer builder
let consumer = TangleConsumer::builder()
    .max_retries(5)
    .initial_backoff(Duration::from_millis(100))
    .max_backoff(Duration::from_secs(30))
    .build();
```

8.3 Chain Reorganizations

The SDK tracks submitted transactions and monitors for inclusion. If a reorg orphans a transaction, the consumer resubmits.

8.4 Crash Recovery

The SDK can checkpoint job processing progress, ensuring restarted operators do not miss or double-process jobs. Enable checkpointing in the runner configuration:

```rust
BlueprintRunner::builder(TangleConfig::default(), env)
    .router(router())
    .producer(TangleProducer::new(service_id))
    .consumer(TangleConsumer::new())
    .checkpoint_dir(env.data_dir.join("checkpoints"))
    .run()
    .await
```

8.5 Graceful Shutdown

On SIGTERM/SIGINT, the SDK completes in-progress jobs, flushes pending submissions, and closes connections before exiting. The runner handles shutdown signals automatically.

9 Quality of Service

The Quality of Service (QoS) package provides monitoring, metrics, and observability for operator services.

9.1 QoS Architecture

The QoS system integrates multiple monitoring backends:

- **Prometheus**: Metrics collection via `/metrics` endpoint
- **OpenTelemetry**: Exports metrics to OTLP-compatible backends (Grafana Cloud, Datadog)
- **Loki**: Structured logging with labels for log querying
- **Grafana**: Dashboards showing throughput, latency, errors, and resources

9.2 Built-in Metrics

- **System metrics**: CPU utilization, memory, disk I/O, network traffic
- **Blueprint metrics**: Jobs executed, execution times, success/failure rates, queue depths
- **Blueprint status**: Uptime, last heartbeat, status codes, status messages

9.3 QoS-Based Slashing

QoS provides data; slashing provides consequences. Connecting these systems enables automatic SLA enforcement.

Architecture: QoS metrics flow from operators to a monitoring service. When metrics violate thresholds, the monitor proposes a slash via Tangle. The service manager's `querySlashingOrigin` hook authorizes the monitor. After dispute window, slashing executes.

9.3.1 Example: Heartbeat-Based Slashing

```
contract HeartbeatSlashingManager {
    uint256 public constant MAX_MISSED = 3;
    uint256 public constant INTERVAL = 5 minutes;

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

    function recordHeartbeat(uint64 serviceId) external {
        require(isOperatorInService(serviceId, msg.sender));
        lastHeartbeat[serviceId][msg.sender] = block.timestamp;
        missedCount[serviceId][msg.sender] = 0;
    }

    function checkAndSlash(uint64 serviceId, address operator) external
        {
        uint256 elapsed = block.timestamp - lastHeartbeat[serviceId][
           operator];
        uint256 missed = elapsed / INTERVAL;

        if (missed > MAX_MISSED) {
            tangle.proposeSlash(serviceId, operator, 500); // 5%
        }
    }

    function querySlashingOrigin(uint64) external view returns (address
       ) {
        return address(this);
    }
}
```

9.3.2 Example: Latency-Based Slashing

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

    address public latencyOracle;
    mapping(uint64 => mapping(address => uint256)) public violations;

    function reportLatencyViolation(
        uint64 serviceId,
        address operator,
        uint256 actualLatencyMs,
        bytes calldata proof
    ) external {
        require(msg.sender == latencyOracle);
        require(actualLatencyMs > MAX_LATENCY_MS);

        violations[serviceId][operator]++;

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

9.3.3 Example: Availability-Based Slashing

```
contract AvailabilitySlashingManager {
    uint256 public constant MIN_UPTIME_PERCENT = 99;
    uint256 public constant WINDOW = 1 days;

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

    function recordCheck(uint64 serviceId, address operator, bool
        available)
        external onlyMonitor
    {
        UptimeRecord storage r = uptime[serviceId][operator];

        if (block.timestamp > r.windowStart + WINDOW) {
            if (r.totalChecks > 0) {
                uint256 pct = (r.successfulChecks * 100) / r.
                    totalChecks;
                if (pct < MIN_UPTIME_PERCENT) {
                    uint256 slash = (MIN_UPTIME_PERCENT - pct) * 100;
                    tangle.proposeSlash(serviceId, operator, uint16(
                        slash));
                }
            }
            r.windowStart = block.timestamp;
            r.successfulChecks = 0;
            r.totalChecks = 0;
        }

        r.totalChecks++;
        if (available) r.successfulChecks++;
    }
}
```

9.4 Connecting SDK to On-Chain Slashing

The full flow:

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

// 2. Off-chain monitor queries metrics
// GET http://operator:9090/metrics

// 3. Monitor detects SLA violation
if avg_latency > SLA_THRESHOLD {
    // 4. Monitor calls service manager
    service_manager.reportLatencyViolation(
        service_id, operator, latency_ms, proof
    ).send().await?;
}
// 5. Service manager proposes slash
// 6. After dispute window, slash executes
```

9.5 Design Considerations

When designing QoS-based slashing:

- Calibrate thresholds to distinguish misbehavior from normal variance
- Ensure measurement integrity through multiple monitors or proofs
- Include evidence in proposals for dispute resolution

- Make penalties proportional to harm
- Consider grace periods for new operators

10 Verifiability and Detection

Slashing requires detection. This section covers verification strategies, worked examples, and formal properties.

10.1 Verification Economics

Not all verification is worthwhile. Let C_v be verification cost, P_d detection probability, S slash amount, and P_c probability of cheating. Verification is justified when:

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

This reveals design principles. High slashes enable profitable verification even with moderate detection. Probabilistic verification (random sampling) achieves deterrence at lower cost. Verification costs should scale with stakes.

10.2 Verification Approaches

Redundant Execution: Multiple operators compute independently; disagreements trigger investigation. Simple but expensive. Use commit-reveal to prevent copying.

Optimistic with Fraud Proofs: Assume honest, challenge suspicious results with cryptographic proofs. Efficient for verifiable computations.

Challenge-Response: Periodic challenges test operator honesty. Maintain database of (input, expected output) pairs indistinguishable from regular jobs. Cost-effective for ongoing services.

TEE Attestation: Trusted Execution Environments provide hardware-backed verification. Strong guarantees but requires TEE hardware.

Reputation and Sampling: Aggregate customer reports. Random sampling selects executions for intensive verification.

10.3 AI-Specific Challenges

AI outputs are often non-deterministic and subjective.

Model Identity: Different models have different output distributions. Fingerprinting probes produce distinctive outputs. Comparing operator distributions against known fingerprints identifies which model runs.

Performance: Claims about tokens/second or latency can be tested empirically. Continuous random sampling prevents inflated benchmarks during known tests.

Isolation: Isolation failures may leave no visible trace. Defense in depth (network policies, syscall filtering, process isolation) makes breaches unlikely. Audit logging and anomaly detection increase detection probability.

10.4 Worked Example: Verifiable AI Sandbox

A sandbox service promising agents run on Claude with specified isolation.

Verification mechanisms:

- Model fingerprinting: Challenge database of (prompt, expected pattern) pairs distinguishing Claude from alternatives
- Performance: Heartbeat jobs with latency requirements

- Isolation: gVisor/Firecracker required at registration; QoS monitors for anomalies

Slashing configuration:

- Model substitution: 50% (severe, violates core promise)
- Performance degradation: 5% (moderate, affects quality)
- Isolation breach: 100% (maximum, compromises security)

Quantified economics. Customer runs trading agents managing \$200,000. PfC = \$200,000 (assets under management).

Service uses 3 operators, each staking \$100,000 with 80% exposure. Corrupting requires all 3 (any honest operator detects substitution via challenges). CoC = $3 \times \$100,000 \times 0.8 = \$240,000$.

Security ratio = \$240,000 / \$200,000 = 1.2. Exceeds 1 but below recommended 1.5x.

Detection probability. Model fingerprinting challenges submitted 10 times daily. Each detects substitution with 85% probability. Over 7-day dispute window:

$$P_d = 1 - (1 - 0.85)^{70} \approx 99.99\%$$

Effectively certain detection justifies the 1.2 ratio, expected cost is $0.9999 \times \$240,000 \approx \$240,000$.

10.5 Worked Example: Verifiable Oracle

A price oracle promising freshness (5 min), accuracy (1%), and availability (99%).

Verification mechanisms:

- Source verification: Signed data from 3+ of 5 approved sources submitted with prices
- Cross-operator comparison: Multiple operators submit independently; deviations >0.5% flagged
- External auditing: Anyone compares oracle prices to public market data

Slashing configuration:

- Source forgery: 100% (complete integrity compromise)
- Price deviation: 10% per incident, accumulating
- Availability failure: 1% per missed update, capped at 20%

Economics. Oracle serves DeFi with \$100M exposure. Manipulated prices enable \$10M arbitrage (PfC = \$10M).

5 operators each staking \$3M with 100% exposure. Corrupting 3-of-5 yields CoC = \$9M.

Security ratio = 0.9, insufficient. Blueprint should increase stake requirements or reduce exposure to smaller protocols.

10.6 Formal Verification Properties

Property V1 (Detection Soundness): If operator behaves honestly, verification flags them for slashing with probability at most ϵ (false positive rate).

Importance: False positives discourage participation. High false positive rate is unusable.

Property V2 (Detection Completeness): If operator violates specification in manner M, verification detects with probability at least p_M.

Importance: Detection probability determines effective cost of cheating.

Property V3 (Economic Sufficiency): For all violation types M with benefit V_M: $p_M \times S_M > V_M$.

Importance: Core security condition. Blueprints must configure detection and slashing to satisfy this.

Property V4 (Verification Efficiency): Verification cost $C_v < \alpha \times V_s$ where V_s is service value.

Importance: Verification costing more than service provides is irrational. Target $\alpha < 0.1$.

Blueprints should analyze mechanisms against these properties, documenting false positive rates, detection probabilities, and costs.

11 Working with the Marketplace

Blueprints interact with the Tangle marketplace for service discovery and pricing. Understanding this interaction helps developers design robust services.

11.1 The RFQ System

Tangle uses request-for-quote (RFQ) for service pricing. Customers broadcast quote requests; operators return signed quotes; customers select and submit.

This accommodates operator heterogeneity, different hardware, locations, and capabilities command different prices.

11.2 Off-Chain Coordination

Quote propagation uses gossip protocols. Customers publish requests specifying blueprint, desired operators, parameters, and deadline. Requests propagate through the P2P network to registered operators.

Direct RPC queries bypass gossip for known operators. Each operator's endpoint is stored on-chain at registration.

Quote aggregators may collect quotes on behalf of customers, presenting curated options.

11.3 Failure Modes

Your blueprint should handle these scenarios:

Operator unavailable after quoting: Operator signs quote but becomes unavailable. Service creates but operator cannot fulfill. Handle through heartbeat monitoring and slashing.

Quote expiry during transaction: Quotes expire before transaction inclusion. Transaction fails with `QuoteExpired`. Set appropriate expiry times (not too short, not too long).

Capacity exhaustion: Operator signs multiple quotes exceeding capacity. Operator's responsibility to track outstanding quotes. Overcommitted operators face quality degradation and slashing.

Network partition: P2P partition prevents reaching some operators. Quote collection fails or produces incomplete results. Recovery occurs when partition heals.

11.4 Quote Serving Architecture

Operators run a separate **pricing engine** service alongside their blueprint. The pricing engine:

1. Receives quote requests via RPC or P2P gossip
2. Calculates prices based on resource requirements and operator pricing models
3. Signs quotes using EIP-712 typed data signatures
4. Returns signed quotes with proof-of-work to prevent spam

The quote contains: blueprint ID, TTL (blocks), total cost, timestamp, expiry, and security commitments (which assets back the operator's stake exposure).

Customers collect signed quotes from multiple operators, compare prices and terms, then submit their chosen quote on-chain to create a service.

11.5 Price Discovery

Prices converge as operators observe market conditions. On-chain service creation reveals accepted prices. Operators analyze historical prices and adjust quotes.

Quality differentiation prevents pure price competition. Operators with better reliability, faster hardware, or stronger security can command premiums.

12 Testing and Development

12.1 Local Testing

Mock producers generate test job calls without blockchain infrastructure:

```
#[tokio::test]
async fn test_square_handler() {
    let result = square(TangleArg(5)).await;
    assert_eq!(result.0, 25);
}
```

12.2 Integration Testing

Use local blockchain instances (Anvil, Hardhat) with deployed contracts:

```
#[tokio::test]
async fn test_full_flow() {
    let anvil = Anvil::new().spawn();
    let contracts = deploy_contracts(&anvil).await;

    // Submit job
    let job_id = submit_job(&contracts, 42).await;

    // Process with blueprint
    let runner = create_test_runner(&contracts).await;
    runner.process_pending().await;

    // Verify result
    let result = get_result(&contracts, job_id).await;
    assert_eq!(result, 42 * 42);
}
```

12.3 Testnet Deployment

Tangle testnet provides realistic pre-production validation with test tokens.

12.4 Observability

The SDK supports:

- Structured logging (configurable verbosity)
- Prometheus metrics export
- OpenTelemetry distributed tracing

13 P2P Networking

The SDK includes peer-to-peer networking built on libp2p, enabling operators to coordinate without centralized infrastructure.

13.1 Network Architecture

The networking layer provides:

NetworkService: Core service managing peer connections, message routing, and protocol handlers. Configured via `NetworkConfig` specifying listen addresses, bootstrap peers, and protocol settings.

NetworkServiceHandle: Thread-safe handle for interacting with the network from job handlers. Supports sending messages, subscribing to topics, and querying peer state.

13.2 Discovery Mechanisms

Kademlia DHT provides global peer discovery. Operators register in the distributed hash table, enabling any participant to find peers by their address or service membership. Bootstrap nodes seed initial peer connections.

mDNS enables local network discovery for development and testing. Operators on the same LAN discover each other automatically without external infrastructure.

PeerManager tracks peer metadata: public keys, addresses, connection state, and service registrations. The manager handles reconnection logic and peer scoring.

13.3 Messaging Protocols

Gossipsub provides efficient pub/sub messaging. Topics are identified by strings (e.g., service IDs). Messages propagate through the mesh network with configurable redundancy. Use cases include:

- Quote request broadcasting
- BLS signature aggregation
- Service announcements

Request-Response Protocol enables direct peer communication. The `BlueprintProtocol` defines typed messages:

- `InstanceMessageRequest`: Authenticated requests between operators
- `InstanceMessageResponse`: Responses with optional error handling

13.4 Authentication and Security

Peer authentication uses cryptographic handshakes:

AllowedKeys configures which peers may connect:

- `EvmAddresses`: Allow peers whose keys derive to specified addresses
- `InstancePublicKeys`: Allow specific public keys

Handshake protocol verifies peer identity before message exchange. Failed handshakes emit `HandshakeFailed` events for monitoring.

13.5 Network Events

The service emits typed events for application handling:

```rust
pub enum NetworkEvent<K: KeyType> {
    InstanceRequestInbound { peer: PeerId, request:
       InstanceMessageRequest<K> },
    InstanceResponseInbound { peer: PeerId, response:
       InstanceMessageResponse<K> },
    GossipReceived { source: PeerId, topic: String, message: Vec<u8> },
    PeerConnected(PeerId),
    PeerDisconnected(PeerId),
    HandshakeCompleted { peer: PeerId },
    HandshakeFailed { peer: PeerId, reason: String },
}
```

Job handlers can subscribe to these events for coordination logic.

13.6 BLS Signature Aggregation

The `agg-sig-gossip` feature provides specialized gossip for BLS signature aggregation. When enabled:

1. Operators compute results and sign with BLS keys
2. Signatures broadcast via gossip topic for the job
3. Aggregator collects signatures until threshold met
4. Aggregated result submitted to chain

This pattern enables efficient multi-operator consensus without centralized coordination.

13.7 Custom Protocols

Blueprints can define custom protocols for domain-specific needs:

- Checkpoint synchronization for stateful services
- Shard assignment coordination for distributed processing
- Heartbeat protocols for liveness monitoring
- State machine replication for consensus-critical services

Custom protocols implement the libp2p `NetworkBehaviour` trait and integrate with the SDK's event system.

14 Deployment Checklist

Before deploying to production:

1. **Testing**: Unit tests, integration tests, testnet validation
2. **Error handling**: All failure modes covered with appropriate responses
3. **Monitoring**: Logging, metrics, and alerts configured
4. **Security**: Input validation, rate limiting, access controls
5. **Documentation**: Operator guide, API reference, troubleshooting
6. **Economics**: Slashing conditions, stake requirements, pricing model
7. **Verification**: Detection mechanisms appropriate to threat model

15 Operator Configuration

Operators configure their blueprint environment through command-line arguments, environment variables, or configuration files.

15.1 Core Settings

http_rpc_url HTTP RPC endpoint for the host chain (default: `http://127.0.0.1:9944`)

ws_rpc_url WebSocket RPC endpoint for event subscriptions (default: `ws://127.0.0.1:9944`)

keystore_uri Path to the keystore directory containing operator keys (default: `./keystore`)

data_dir Directory for blueprint-specific data storage (required)

protocol Target protocol: `tangle`, `eigenlayer`, or `symbiotic`

chain Network: `local_testnet`, `testnet`, or `mainnet`

15.2 Networking Settings

bootnodes Bootstrap peer addresses for P2P discovery (multiaddr format)

network_bind_port Port for P2P connections (default: random)

enable_mdns Enable local network discovery for development (default: false)

enable_kademlia Enable DHT-based global discovery (default: true in production)

target_peer_count Target number of peer connections (default: 24)

15.3 Example Configuration

Command-line invocation:

```
./my-blueprint run \
    --http-rpc-url https://rpc.tangle.tools \
    --ws-rpc-url wss://rpc.tangle.tools \
    --keystore-uri /var/lib/blueprint/keystore \
    --data-dir /var/lib/blueprint/data \
    --protocol tangle \
    --chain mainnet \
    --bootnodes /ip4/34.56.78.90/tcp/9000/p2p/12D3Koo... \
    --network-bind-port 9000 \
    --enable-kademlia
```

Environment variables use uppercase with underscores:

```
export HTTP_RPC_URL=https://rpc.tangle.tools
export WS_RPC_URL=wss://rpc.tangle.tools
export KEYSTORE_URI=/var/lib/blueprint/keystore
export DATA_DIR=/var/lib/blueprint/data
export PROTOCOL=tangle
export CHAIN=mainnet
```

15.4 Keystore Setup

The keystore contains cryptographic keys for operator identity and signing:

```
# Generate required keys
blueprint-cli key generate --scheme ecdsa --output ./keystore
blueprint-cli key generate --scheme ed25519 --output ./keystore
blueprint-cli key generate --scheme bls --output ./keystore

# List keys
blueprint-cli key list --keystore ./keystore
```

Required key types depend on the blueprint:

- **ECDSA**: Transaction signing and EVM address derivation
- **Ed25519**: P2P network identity and authentication
- **BLS**: Signature aggregation for multi-operator consensus

16 Resources

- **SDK Repository**: `https://github.com/tangle-network/blueprint-sdk`
- **API Documentation**: `https://docs.tangle.tools/sdk`
- **Example Blueprints**: `https://github.com/tangle-network/blueprints`
- **Discord**: `https://discord.gg/tangle`