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 is 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 via 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 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:

- `TangleEvmProducer` watches blockchain events
- `CronProducer` 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_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
}
```

3.3 Understanding the Code

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
- `TangleEvmLayer` handles encoding/decoding

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.

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

`TangleEvmProducer` 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 arbitrary trigger conditions, implement `BackgroundKeeper`:

```
struct PriceMonitorKeeper;

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

    fn start(config: KeeperConfig, mut shutdown: 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 conditions and trigger actions
        Ok(false)
    }
}
```

Built-in keepers include:

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

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
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
    })
}
```

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
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| Error::Other(e.to_string()))?;

    TangleEvmResult(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.

```rust
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.

```rust
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(|| Error::Other("No DKG share".into()))?;
    let partial = key_share.partial_sign(&message)?;

    TangleEvmResult(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.

8.1 Handler Errors

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

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.

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 don't miss or double-process jobs.

8.5 Graceful Shutdown

On SIGTERM/SIGINT, the SDK completes in-progress jobs, flushes pending submissions, and closes connections before exiting.

9 Verifiability and Detection

Slashing requires detection. This section covers verification strategies.

9.1 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:

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

9.2 Verification Approaches

Redundant Execution: Multiple operators compute independently; disagreements trigger investigation. Simple but expensive.

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

Challenge-Response: Periodic challenges test operator honesty. Cost-effective for ongoing services.

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

9.3 AI-Specific Challenges

AI outputs are often non-deterministic and subjective. Approaches include:

- Model fingerprinting (distinguishing models by output patterns)
- Semantic consistency checking
- Challenge databases with known-correct answers
- Ensemble comparison across operators

10 Testing and Development

10.1 Local Testing

Mock producers generate test job calls without blockchain infrastructure:

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

10.2 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);
}
```

10.3 Testnet Deployment

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

10.4 Observability

The SDK supports:

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

11 P2P Networking

The SDK includes peer-to-peer networking built on libp2p.

11.1 Discovery

Kademlia DHT for global peer discovery.
mDNS for local network discovery.

11.2 Messaging

Gossipsub for efficient pub/sub message propagation.
Direct connections for point-to-point coordination.

11.3 Custom Protocols

Blueprints can define custom protocols for:

- Checkpoint synchronization
- State sharing
- Domain-specific coordination

12 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

13 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