ANASTASIA LABS

Lucid Evolution 2.0

Proof of Achievement - Milestone 1

Project Id	1300128
Project Manager	Jonathan Rodriguez
Proposal Link	Catalyst Proposal

Contents

Private Testnet SDK & L2 Provider Integration Design Specification Document

1. Executive Summary

Lucid Evolution 2.0 - Private Testnet SDK & L2 Provider Integration introduces a powerful **TypeScript SDK** designed to provide Cardano developers with a **reliable and efficient testing environment** for smart contracts and decentralized applications. It directly tackles the common frustrations of current development workflows, such as reliance on incomplete emulators, cumbersome shell scripts, and manual Docker commands. This SDK transforms the experience by offering developers **programmable control over real Cardano nodes and integrated Hydra heads**.

A key benefit is the ability to **spin up on-demand, disposable private testnets in under 30 seconds** using technologies like Dockerode and Testcontainers, all powered by a concise **Effect-TS API**. This fundamentally shifts infrastructure control into the codebase, eliminating external dependencies and enabling seamless integration with popular testing frameworks. Ultimately, Lucid Evolution 2.0 empowers developers to **focus on application logic rather than infrastructure management**, significantly enhancing productivity and reliability.

1.1. Purpose of this Document

This Design Specification Document **outlines the architecture**, **functionalities**, and **user requirements** for the Lucid Evolution 2.0 Private Testnet SDK. It serves as a foundational document for the **development team**, **stakeholders**, and **community**, ensuring a shared understanding of the SDK's scope, design principles, and expected behavior.

2. Problem Analysis and Solution Rationale

This section details the challenges in current Cardano development workflows and explains the strategic reasoning behind the Lucid Evolution 2.0 SDK.

2.1. Identified Problems

Current Cardano development workflows face several significant obstacles:

- **External Tool Dependencies:** Developers often rely on bash scripts or raw Docker commands that are separate from their main testing code. This leads to unstable setups and inconsistent test results.
- **Inaccurate Environments:** Many existing testing tools use incomplete emulators that do not accurately simulate real Cardano blockchain behavior. This causes unreliable tests that fail to mirror mainnet conditions.
- **Complex Setup:** Manual configuration of private testnets requires extensive infrastructure knowledge and consumes significant developer time, delaying project initiation.
- **Poor Integration:** The lack of direct TypeScript integration with testing tools creates workflow inefficiencies, requiring developers to manage disparate systems.

2.2. Design Objectives

Our solution architecture is designed to overcome these problems by achieving the following core objectives:

- **Direct Control:** Provide intuitive TypeScript APIs for complete testnet lifecycle management.
- **Real Node Integration:** Utilize actual Cardano nodes instead of emulators to ensure authentic blockchain behavior and accurate testing.
- **Enhanced Developer Experience:** Offer simple default configurations for quick starts, alongside comprehensive customization options for advanced use cases.
- **Production Parity:** Ensure that private testnets behave identically to the Cardano mainnet, enabling highly reliable and confident contract testing.

2.3. Key Advantages

This SDK offers substantial advantages throughout the development lifecycle:

- **80% Reduction** in developer setup time for testing environments.
- **Elimination of external script dependencies**, simplifying workflows and reducing maintenance.
- **Enhanced Reliability** through real Cardano node integration, leading to more accurate test results.
- **Improved Test Coverage** via consistent, reproducible environments, fostering higher quality dApps.

3. Architecture Overview

This section presents a high-level overview of the SDK's architecture, illustrating how its core components interact to provide private testnet functionality.

The diagram above illustrates the layered architecture of the Lucid Evolution 2.0 SDK. Here's a breakdown of each key component:

- **DevNet Management Layer:** This part of the SDK handles the full lifecycle of your testnets. It manages container orchestration (starting, stopping, and removing them), generates necessary configurations (like genesis files and cryptographic keys), and ensures proper resource and port isolation for each testnet.
- **Provider & L2 Integration Layer:** This layer connects your application to the testnet. It implements Lucid's Provider interface, letting you submit transactions, query UTxOs, retrieve datums, and interact with the blockchain. It also extends this functionality to manage Layer 2 solutions like Hydra heads, off-chain transactions, and on-chain settlement processes.
- **Container Runtime Libraries (Dockerode / Testcontainers):** These are Node.js libraries that provide the programming interface to interact directly with Docker. They allow the SDK to programmatically control Docker containers.
- **Underlying Infrastructure (Docker Runtime):** This is the foundation. It's the Docker engine running real Cardano nodes (supporting any era) and Hydra nodes, each isolated within its own container. This ensures your tests run against environments that truly mimic the production network.

3.1. Technology Stack

The technology stack for Lucid Evolution 2.0 was selected to ensure robustness, type safety, and direct control over the testing environment.

3.1.1. Effect-TS

- **Decision:** We adopted Effect-TS for functional programming and robust error handling.
- **Rationale:**
  1. **Composability:** Complex container operations are composed functionally, resulting in modular and reusable code.
  2. **Error Safety:** Tagged Errors provide precise failure context and structured recovery strategies, improving debugging and resilience.
  3. **Type Safety:** Comprehensive static type checking eliminates common runtime errors, leading to more stable code.
  4. **Concurrency** Offers superior handling of concurrent container operations, crucial for efficient testnet management.

- **Implementation Example:**

```
export class CardanoDevNetError extends Data.TaggedError("CardanoDevNetError")<{
  reason: "container_not_found" | "container_creation_failed" | "container_start_failed" |
          "container_stop_failed" | "container_removal_failed" |
"container_inspection_failed" |
          "temp_directory_creation_failed" | "config_file_write_failed" |
"file_permissions_failed";
  message: string;
  cause?: unknown;
}> {}
```

3.1.2. Dockerode Integration

- **Decision:** We utilize **Dockerode** for direct programmatic interaction with the Docker API.
- **Rationale:**
  1. **Programmatic Control:** Provides direct API access, eliminating reliance on brittle shell commands.
  2. **Structured Error Handling:** Enables structured error responses from Docker, allowing for precise failure diagnosis and recovery.
  3. **Resource Management:** Offers fine-grained control over container lifecycle and resources, optimizing utilization.
  4. **Cross-Platform Consistency:** Ensures uniform behavior across macOS, Linux, and Windows environments (via WSL2).

3.1.3. Real Cardano Node Approach

- **Decision:** We deploy actual Cardano nodes within Docker containers instead of emulators.
- **Rationale:**
  - **Authentic Behavior:** Guarantees identical transaction processing and protocol semantics to production networks.
  - **Full Protocol Compliance:** Provides complete support for all Cardano eras and features, ensuring comprehensive test coverage.
  - **Future Compatibility:** Automatically adapts to protocol upgrades, reducing maintenance overhead.

- **Accurate Debugging:** Real node logs and metrics offer authentic insights for troubleshooting.

3.2. Configuration System Design

A flexible and secure configuration system is essential for adapting testnets to diverse development needs.

3.2.1. Comprehensive Era Support

- **Decision:** We support configuration for all Cardano eras (Byron, Shelley, Alonzo, Conway).
- **Rationale:**
  - **Historical Compatibility:** Allows testing of legacy transaction formats and behaviors.
  - **Era Transition Testing:** Enables validation of applications across hard fork boundaries.
  - **Complete Protocol Coverage:** Ensures support for all Cardano features and capabilities.
  - **Future Proofing:** The design is extensible for upcoming eras.
- **Structure Example:**

```
interface DevNetConfig {
  readonly containerName?: string;
  readonly image?: string;
  readonly ports?: { readonly node: number; readonly submit: number };
  readonly networkMagic?: number;
  readonly nodeConfig?: Partial<NodeConfig>;
  readonly byronGenesis?: Partial<ByronGenesis>;
  readonly shelleyGenesis?: Partial<ShelleyGenesis>;
  readonly alonzoGenesis?: Partial<AlonzoGenesis>;
  readonly conwayGenesis?: Partial<ConwayGenesis>;
  readonly kesKey?: Partial<KesKey>;
  readonly opCert?: Partial<OpCert>;
  readonly vrfSkey?: Partial<VrfSkey>;
}
```

3.2.2. Security-First Key Management

- **Decision:** We implement secure cryptographic key handling with proper file permissions.
- **Rationale:**
  - **Security Best Practices:** File permissions (0o600) prevent unauthorized access to sensitive keys.
  - **Ephemeral Keys:** Temporary keys reduce the window of security exposure.
  - **Automatic Cleanup:** Secure destruction of keys prevents leakage after testing.
  - **Isolation:** Per-testnet keys prevent cross-contamination between environments.

3.3. Multi-Instance Architecture

The SDK is designed to support multiple, independently operating testnet instances, enabling parallel testing and complex simulation scenarios.

3.3.1. Isolation Strategy

- **Decision:** We ensure complete isolation between testnet instances.
- **Rationale:**
  - **Parallel Testing:** Enables multiple test suites to run simultaneously without interference.
  - **Resource Safety:** Prevents resource conflicts and port collisions between instances.
  - **State Isolation:** Maintains independent blockchain states for different test scenarios.
  - **Failure Containment:** Issues in one testnet instance do not affect others.

3.3.2. Resource Management

- **Decision:** We provide configurable resource limits with intelligent defaults.
- **Rationale:**
  - **System Protection:** Prevents resource exhaustion on developer machines.
  - **Predictable Performance:** Ensures consistent behavior across different hardware configurations.
  - **Cost Efficiency:** Optimizes resource utilization for development workflows.
  - **Scalability:** Supports both single and multi-instance scenarios efficiently.

4. Core Component Design

This section details the primary internal components of the SDK, describing their functions and design benefits.

4.1. Container Management Layer

This foundational layer handles all interactions with Docker, overseeing the entire lifecycle of Cardano node containers. It's responsible for the direct programmatic control over your DevNet instances.

- **Core Functions:** This layer exposes the following key functions for managing DevNet containers:
  - **`make(config?: DevNetConfig):`** Creates and provisions a new DevNet container, making it ready for use.
  - **`startContainer(container: DevNetContainer):`** Starts a previously created or stopped DevNet container.
  - **`stopContainer(container: DevNetContainer):`** Stops a running DevNet container without removing its data.
  - **`removeContainer(container: DevNetContainer):`** Removes a DevNet container and its associated data, cleaning up resources.
  - **`getContainerStatus(container: DevNetContainer):`** Retrieves the current status and detailed information about a specific DevNet container.

- **Primary Functions:**
  - **Lifecycle Operations:** Manages container creation, starting, stopping, and removal with comprehensive error handling.
  - **Configuration Generation:** Dynamically generates genesis files and node configurations within secure temporary directories.
  - **Port Management:** Automatically allocates and manages ports for node communication, including collision detection.
  - **Volume Mounting:** Securely mounts configuration and key files into containers.

- **Design Benefits:**
  - **Reliability:** Ensures robust operation with comprehensive error handling and clear error messages.

- **Security:** Implements proper file permissions and secure key management practices.
- **Flexibility:** Allows for customizable configuration with sensible defaults.
- **Maintainability:** Promotes clear separation of concerns and a modular design.

4.2. Provider Integration Layer

This component bridges the DevNet functionality with Lucid's API and supports Layer 2 protocols like Hydra.

- **Integration with Lucid Evolution:**

  Implements Lucid's Provider type so DApps written with Lucid can point to a DevNet instance transparently.

  Example:

```
  import { fromDevNet } from "@lucid-evolution";
  import { make, startContainer } from "@lucid-evolution/experimental/CardanoNode/
DevNet";

  const main = Effect.gen(function* () {
    const container   = yield* make({ networkMagic: 42 });
    yield* startContainer(container);
    const provider = fromDevNet(container, { networkMagic: 42 });
    const lucid       = Lucid.new(provider, { Key });
    // ... use lucid to submit transactions, query UTxOs, etc.
  });
```

- **Transparent Failover:** If a DevNet container stops unexpectedly, DevNetProvider will automatically attempt to restart it (configurable in DevNetConfig).

4.3. L2 (Hydra) Integration Layer

This layer will provide comprehensive tools for integrating and managing Hydra heads with your DevNet.

4.3.1. Hydra Head Lifecycle

- **Head Initialization (initHead):** Spawns a Hydra leader container alongside the Cardano node, sets up network ports for head communication.
- **Open/Close/Fanout:** Exposes TypeScript methods to open state channel heads, submit off-chain transactions, and coordinate on-chain settlement via L1.
- **Participant Management:** Add/Remove peers in a head; track on-chain UTxOs for deposits and snapshot states.

4.3.2. Off-Chain Transaction Pipeline

- **Stability Guarantees:** Ensures off-chain transactions build and sign correctly against the DevNet node's ledger state.
- **Event Subscriptions:** Expose RxJS or Effect-TS streams to notify developers when head state changes, on-chain commits, or contests occur.
- **Automated Settlement:** On test failure or head closure, automatically coordinate Hydra head on-chain transactions using the DevNet provider.

5. Error Handling Strategy

A robust error handling system is critical for a reliable SDK, ensuring stability and a positive developer experience.

5.1. Tagged Error System

- **Design Philosophy:** Precise error categorization using Effect-TS's Tagged Errors enables targeted recovery strategies.
- **Implementation Benefits:**
  - **Debugging Efficiency:** Provides clear error types with actionable remediation steps for developers.
  - **Automated Recovery:** Allows for specific error handlers for different failure modes, enabling programmatic recovery.
  - **User Experience:** Generates helpful error messages with troubleshooting guidance.
  - **System Reliability:** Facilitates graceful degradation and automatic cleanup in case of failures.

5.2. Operational Resilience

The SDK is designed with built-in mechanisms to recover from unexpected issues and maintain system stability.

- **Recovery Mechanisms:**
  - **Container Cleanup:** Automatic removal of failed or terminated containers.
  - **Resource Reclamation:** Cleanup of temporary files and directories.
  - **State Recovery:** Intelligent restart and state restoration capabilities.
  - **Graceful Degradation:** Ensures partial functionality during component failures.

6. Key Interfaces & Types

This section details the essential TypeScript interfaces and types that developers will interact with when using the SDK, providing a clear contract for its functionality.

Error Handling Definition

```
export class CardanoDevNetError extends Data.TaggedError("CardanoDevNetError")<{
  reason: "container_not_found" | "container_creation_failed" | "container_start_failed" |
          "container_stop_failed" | "container_removal_failed" |
"container_inspection_failed" |
          "temp_directory_creation_failed" | "config_file_write_failed" |
"file_permissions_failed";
  message: string;
  cause?: unknown;
}> {}
```

DevNet Container Interface

```
  interface DevNetContainer { id: string; name: string; }
```

DevNet Lifecycle Functions

These core functions manage the DevNet's state and return an Effect type for robust error handling.

```
  export const make: (
    config?: DevNetConfig
  ) => Effect<DevNetContainer, CardanoDevNetError>

  export const startContainer: (
    container: DevNetContainer
  ) => Effect<void, CardanoDevNetError>

  export const stopContainer: (
    container: DevNetContainer
  ) => Effect<void, CardanoDevNetError>

  export const removeContainer: (
    container: DevNetContainer
```

```
  ) => Effect<void, CardanoDevNetError>

  export const getContainerStatus: (
    container: DevNetContainer
  ) => Effect<Docker.ContainerInspectInfo, CardanoDevNetError>
```

DevNet Configuration Interface

This comprehensive interface allows for deep customization of the DevNet.

```
export interface DevNetConfig {
  readonly containerName?: string;
  readonly image?: string;
  readonly ports?: {
    readonly node: number;
    readonly submit: number;
  };
  readonly networkMagic?: number;
  readonly nodeConfig?: Partial<NodeConfig>;
  readonly byronGenesis?: Partial<ByronGenesis>;
  readonly shelleyGenesis?: Partial<ShelleyGenesis>;
  readonly alonzoGenesis?: Partial<AlonzoGenesis>;
  readonly conwayGenesis?: Partial<ConwayGenesis>;
  readonly kesKey?: Partial<KesKey>;
  readonly opCert?: Partial<OpCert>;
  readonly vrfSkey?: Partial<VrfSkey>;
}
```

Genesis Configuration Types

Detailed types for each Cardano era's genesis file (**`NodeConfig`**, **`ByronGenesis`**, **`ShelleyGenesis`**, **`AlonzoGenesis`**, **`ConwayGenesis`**) and cryptographic key components (KesKey, OpCert, VrfSkey) enable granular control over network parameters.

Default Configuration Constants

Pre-defined default values for each configuration type (e.g., **`DEFAULT_NODE_CONFIG`**, **`DEFAULT_BYRON_GENESIS`**, **`DEFAULT_DEVNET_CONFIG`**) ensure zero-config usability.

Provider Integration Interface

This interface ensures compatibility with Lucid's existing **`Provider`** standard, allowing dApps to interact with the DevNet seamlessly.

```
interface DevNetProvider extends Provider {
  readonly url: string; // e.g., "http://localhost:9232"
  readonly networkMagic: number;
  readonly container: DevNetContainer;

  submitTx(tx: Transaction): Promise<TxHash>;
  awaitTx(txHash: TxHash, timeout?: number): Promise<boolean>;
  getUtxos(addressOrCredential: Address | Credential): Promise<UTxO[]>;
  getUtxosWithUnit(addressOrCredential: Address | Credential, unit: Unit):
Promise<UTxO[]>;
  getUtxoByUnit(unit: Unit): Promise<UTxO>;
  getUtxosByOutRef(outRefs: OutRef[]): Promise<UTxO[]>;
  getDelegation(rewardAddress: RewardAddress): Promise<Delegation>;
  getDatum(datumHash: DatumHash): Promise<Datum>;
  getProtocolParameters(): Promise<ProtocolParameters>;
}
```

7. User Requirements

This section details the perspectives of various stakeholders and the specific capabilities (Functional Requirements) and performance, resource, and compatibility characteristics (Non-Functional Requirements) that the SDK must meet.

7.1. Stakeholder Profiles

- **DApp Developers:** Need quick, reliable private testnets for unit/integration tests.
- **Smart Contract Auditors:** Require deterministic environments to reproduce and debug on-chain behavior.
- **Project Managers:** Seek clear timelines, documented architecture, and assurance of production parity.
- **Community Contributors:** Demand modular design to extend L2 support or add new utilities.

7.2. Functional Requirements

- **FR-001: DevNet Creation:** Users can call make(config?) to provision a new DevNet container with default or overridden settings.
- **FR-002: DevNet Startup:** Users can call startContainer(container) and await readiness (node sync).
- **FR-003: DevNet Teardown:** Users can stop and remove a container via stopContainer(container) and removeContainer(container).
- **FR-004: Transaction Submission:** Via DevNetProvider.submitTx(tx), users can submit serialized transactions to the testnet.
- **FR-005: UTxO and Datum Queries getUtxos:**, getDatum, and related methods return on-chain data.
- **FR-006: Protocol Parameter Retrieval:** Users can fetch current protocol parameters via getProtocolParameters().
- **FR-007: Hydra Head Operations:** Users can initialize, open, close, and fanout Hydra heads using TypeScript APIs.

- **FR-008: Custom Genesis Configuration:** Users can supply custom partial genesis objects for each era.
- **FR-009: Multi-Instance Support:** The SDK must allow parallel DevNets with isolated resources and no port collisions.
- **FR-010: Testing Framework:** Integration Provide Jest/Vitest matchers and utilities for automated assertions.
- **FR-011: Error Reporting All:** container and provider errors propagate as CardanoDevNetError with reason and message.
- **FR-012: Resource Limits:** Default CPU and memory caps per container; configurable in DevNetConfig.

7.3. Non-Functional Requirements

- **Performance:**
  - **Startup Time: Under 30 seconds** from `make()` to node readiness (socket available).
  - **Concurrent Instancing:** Support over 20 simultaneous DevNets on a 16 GB RAM, 4-core machine.

- **Reliability & Resilience:**
  - **Uptime:** 99.9% for container operations in local development contexts.
  - **Auto-Recovery:** Optional automatic restart attempts if a node or Hydra head fails.
  - **Error Handling:** Graceful degradation—non-blocking errors (e.g., optional Hydra) do not crash the SDK.

- **Usability:**
  - **Zero-Config Defaults:** Reasonable defaults for DevNetConfig (networkMagic: 42, ports: {node: 4001, submit: 8090}, image: official Cardano Node).
  - **Type Safety:** Complete TypeScript definitions with IntelliSense.
  - **Documentation:** Comprehensive code comments, usage examples, and a Getting Started guide.

- **Compatibility & Portability:**
  - **Operating Systems:** macOS (with Docker Desktop), Linux (x86_64), Windows 10/11 (with WSL2).

- **Node.js:** Version 16+ (LTS), TypeScript 4.5+.
- **Cardano Node Versions:** Compatible with Cardano Node 10.4.1; plan for backward compatibility to 10.3 and forward to next minor release.
- **Hydra:** Compatible with Hydra node version 1.0+.

8. Feature Prioritization

This table outlines the planned development order based on community feedback and project needs.

9. Conclusion

This design specification documents the complete architecture, functional requirements, and prioritized feature set for Milestone 1 of Lucid Evolution 2.0. The SDK's modular, TypeScript-first approach—powered by Effect-TS and Dockerode/Testcontainers—ensures developers can quickly provision authentic Cardano testnets, including L2 (Hydra) capabilities, with minimal setup.

10. Next Steps

1. Milestone 2 (Core SDK Development): Implement DevNet container layer, basic DevNetProvider, and test utilities (0–2 months).
2. Milestone 3 (Documentation & Onboarding): Publish full developer guides, example repositories, and run community workshops (2–3 months).
3. Milestone 4 (Feature Optimization): Add custom genesis, multi-instance, logging, and retry enhancements; perform extensive performance tuning (3–4 months).
4. Final Milestone (L2 Integration & Release): Complete Hydra head integration, state snapshots, CLI tool, visual explorer; prepare community release and closeout report (4–6 months).

Upon approval, the team will begin Milestone 2, adhering to the roadmap and using the performance/risk metrics defined herein. Successful delivery of this SDK will significantly improve Cardano developer productivity, test coverage, and ecosystem reliability.