Creating new Ledgera applications

A technical introduction

v1

This guide walks through everything needed to build a domain-specific application on top of the Ledgera Byzantine fault-tolerant (BFT) distributed system. It is structured in three parts: an introduction to the essential Ledgera and Rust concepts, a step-by-step walk-through of two reference applications (`app_string_concat` as a minimal hello-world and `app_varkeep` as a full atomic register), and a final section that shows how to turn the blank template into your own application. Each reference application is covered in the same four steps—application design, network vocabulary (`lat_binding.rs`), client-side logic and text-based user interface (TUI), and running the test harness—so the structure is reusable once learned.

Contents

1 Prerequisites

Clone the Git repository at the following url:

`https://github.com/CEA-LIST/Ledgera.git`

then follow the installation instructions in the README. After this step you have:

- **Rust** installed via `rustup`. The exact toolchain is defined in `code/rust-toolchain.toml`; rustup downloads it automatically on the first call to `cargo`.
- The **system dependencies** for compilation, notably `libssl-dev` / `openssl-devel` depending on the distribution (without which `openssl-sys` fails).
- **Python 3**, required to run `launch_test.py` in Step 4.

Quick check:

```
cargo --version
python3 --version
```

2 Essential Ledgera concepts

This section introduces the concepts required to understand and build a Ledgera application. Refer to the Yellow Paper for detailed formal specifications.

Definition

Ledgera is a Byzantine-fault-tolerant (BFT) distributed computing platform running across several nodes called **Ledgents** (Ledgera Agents).

2.1 Roles

Each Ledgent can hold several roles simultaneously:

Client	Entry point through which users and applications interact with Ledgera. Submits function instances and proposes input values for multi-party function instances.
Voter	Co-locates the *Executor* and *Prover* roles from the Yellow Paper. As an Executor, it votes to grant execute rights to function instances, participates in the agreement on unknown input values (multi-party case), executes the replicated function locally, votes on the result, and forwards storage requests for persistent data. As a Prover, it collects quorums of votes and produces the corresponding **Ledgera Proofs**: Proof of Function Declaration, Proof of Unknown Arguments Assignment, Proof of Integrity, and Proof of Storage. It also notifies Clients of these events.
Storer	Stores inputs and outputs of function instances that are flagged as persistent, and answers retrieval queries.
Logger	Totally orders and anchors Ledgera Proofs in a tamper-evident secure log, providing auditability and traceability.

2.2 What is a Ledgera application?

A Ledgera application defines **domain-specific** bindings for what can be computed, stored, and validated on a Ledgera network. Concretely, an application defines the following elements:

Element	Description
Data types	The domain values that flow through computations and can be stored.
Error types	Domain-specific errors that computations can produce.
Computations	Functions that produce an output. Each one must implement a "`ledgera_compute`" function that turns an array of input data into an output data, or fails and returns an error. Each Voter execute this "`ledgera_compute`" function locally to replicate the execution.
Tags	Operations that take inputs but produce no computed output – used to anchor or agree on a set of values without executing a function.
Local predicates	Per-value constraints verified by Voters before accepting an input (e.g. a string must be non-empty).
Global predicates	Constraints over the full input set of a function instance (e.g. two arguments must be distinct).

2.3 Lifecycle of a function instance

The unit of work in Ledgera is a **function instance**: one execution of a Computation or Tag on a concrete set of inputs, identified by a unique client signature. Its lifecycle unfolds as follows:

1. A **Client** submits a function instance via `compute_function(spec)`. The spec bundles the operation, its arguments (concrete values or unknown slots), and the predicates to enforce.
2. Each **Voter** checks the predicates, runs `ledgera_compute` locally on the agreed inputs, and votes on the result digest. A quorum of matching votes produces a **Ledgera Proof of Integrity**.
3. If any input or output is flagged as persistent, **Storers** store it and produce a **Ledgera Proof of Storage**.
4. **Loggers** anchor the proofs in the secure log.
5. The **Client** receives a `ValidatedCoreFeedbackMessage`—signaling that the function instance has been validated by a BFT quorum.

If a function instance has unknown input slots (multi-party case), they are filled later via `propose_input`; Voters wait for agreement on all inputs before executing. Stored data can be retrieved via `retrieve_data(digest)`, and the secure log can be queried via `audit_log(query)`.

2.4 Interacting with a Ledgera application

When the Ledgera Core client node starts up, a `CoreClientRuntime` object is created and returned, bundled with the incoming `ValidatedCoreFeedbackMessage` receiver. The node runner then passes this output to the method running the application / domain-specific logic. From there, this `CoreClientRuntime` ob-

ject is the internal handle through which the application code submits requests to the Ledgera network. It exposes four methods that constitutes its API:

Method	Purpose
`compute_function`	Broadcasts a function instance proposal (RFUN message). The inputs provided to that `compute_function` method include: the operation to execute, its arguments (concrete values or unknown slots), and the predicates to enforce. Then, `compute_function` returns the 64-bit signature that uniquely identifies the submitted instance.
`propose_input`	Broadcasts an input proposal (RIN message) to fill an unknown slot of an already-submitted multi-party function instance.
`retrieve_data`	Queries the Storers for the domain value stored under the given digest. The optional `ProofOfShipmentToStorage` helps locate the data; if absent, the query is sent without proof.
`audit_log`	Queries the secure log for anchored transactions matching the query and returns the list of matching `LedgeraTransaction` values.

In addition to the above outgoing calls, the Core client forwards incoming events to the application layer as `ValidatedCoreFeedbackMessage` values on a multi-producer single-consumer (MPSC) channel (c.f. §3.7). This enum has five variants:

Variant	Meaning
`ValidatedInstance`	The function instance has been declared and granted execute rights. Carries the original request spec and a `ProofOfFunctionDeclaration`.
`Nout`	A computation result notification has been received and validated. It also carries the result value.
`DeliveredTsto`	A `ProofOfShipmentToStorage` has been anchored – data has been durably stored.
`DeliveredTins`	A `ProofOfUnknownArgumentsAssignmentVerification` has been delivered – the unknown inputs of a multi-party instance have been agreed upon.
`DeliveredTout`	A `ProofOfOperationIntegrity` has been anchored – the integrity of the output of a computation has been recorder.

Application (domain-specific) logic must be written to react to these messages.

2.5 Two reference applications

The rest of this guide explains how to build a Ledgera application by walking through two concrete examples:

- `app_string_concat` – the simplest possible application: it concatenates two strings. Its service binding file `lat_binding.rs` is the reference for defining data types, computations, tags, and predicates.
- `app_varkeep` – a distributed variable store (an atomic register). It demonstrates the full client-side structure: service messages, the three `react_to_*` handlers, and the

event loop.

Both are in the `ledgera_code/crates/` folder of this repository.

3 Essential Rust concepts

This section covers the Rust vocabulary that appears throughout a Ledgera application. It is not a complete language tutorial; it focuses on the specific constructs you will encounter when reading or writing a Ledgera application.

3.1 Code organisation

Cargo, crates, workspaces, modules

crate — one Rust library or binary, with one root source file.
Cargo.toml — a crate's manifest: its name and dependencies.
workspace — a collection of crates sharing a root `Cargo.toml`; crates inside can depend on each other by name.
module (`mod`) — a namespace inside a crate; the command `pub mod foo;` exposes `src/foo.rs` as sub-module `foo`.
cargo — Rust's build system and package manager. The command `cargo check` / `cargo build --release` — type-checks/builds a crate.

3.2 Data types

A **struct** groups named fields. Methods are added in a separate `impl` block:

```
pub struct Point { pub x: f64, pub y: f64 }

impl Point {
    pub fn distance(&self, other: &Point) -> f64 { /* ... */ }
}
```

An **enum** defines a type with a fixed set of variants; variants can carry data and are matched exhaustively with `match`:

```
pub enum Msg { Quit, Move { x: i32, y: i32 }, Write(String) }

match msg {
    Msg::Quit         => { /* ... */ }
    Msg::Move { x, y } => { /* use x, y */ }
    Msg::Write(s)     => { /* use s */ }
}
```

Methods can also be added to an **enum** in a separate `imp` block.

Three generic types are often used:

- `Vec<T>` – growable array of elements of type `T` ; `.push(x)`, `.get(i)`, `.len()`.
- `Option<T>` – `Some(value)` of type `T` or `None` (no `null` in Rust).
- `Result<T,E>` – `Ok(value)` of type `T` or `Err(e)` of type `E`; `?` propagates errors.

3.3 Traits

A **trait** defines the methods a type must provide—analogous to a Java interface, or to a Haskell class.

```
// Trait definition (in ledgera-types -- you do not write this):
pub trait LedgeraPublishableMessage {
    fn get_msg_type() -> &'static str;
}

// Your implementation:
impl LedgeraPublishableMessage for MyServiceMsg {
    fn get_msg_type() -> &'static str { "myService/msg1" }
}
```

The keyword sequence is `impl Trait for Type`. The `ledgera_compute` and `ledgera_single_arg_is_valid` methods you write in the service binding file `lat_binding.rs` also satisfy traits; the attribute macros (§3.6) wire those implementations up behind the scenes.

3.4 Trait associated types

A trait can declare **associated types**: named type slots that each implementation fills in concretely, letting the trait express "the data type *this* implementation uses" without extra parameters on every function.

The central Ledgera trait is `LedgeraApplicationTemplate`:

```
pub trait LedgeraApplicationTemplate: Send + Sync + 'static {
    fn get_service_name(&self) -> &str;
    type RuntimeError;
    type Data;
    type Compute;
    type Tag;
    type LocalPredicate;
    type GlobalPredicate;
}
```

This is why Ledgera Core is fully generic: it only depends on a `LAT: LedgeraApplicationTemplate` and accesses `LAT::Data`, `LAT::Compute`, etc., without knowing which application is running. The feedback message you receive in the handler `react_to_core.rs` is therefore typed with your concrete template—`ValidatedCoreFeedbackMessage<YourBackend>`—so the compiler resolves all generic slots to your domain types automatically (see §5.3 for an example).

3.5 Generics and trait bounds

A **generic** type is parameterised by a type variable; a **trait bound** (`T: SomeTrait`) restricts that variable to types that implement the trait, akin to a class constraint in Haskell.

You encounter this in the service client struct, which is generic over the cryptographic backend and the communication layer:

```
pub struct LedgeraServiceClientBehavior<
    PKI: PublicKeyInfrastructure,
    Sess: PubSubNetwork
> { /* ... */ }

impl<
    PKI: PublicKeyInfrastructure,
    Sess: PubSubNetwork
> LedgeraServiceClientBehavior<PKI, Sess> {
    pub async fn react_to_validated_core_msg(&mut self, /* ... */)
```

```
        { /* ... */ }
}
```

You copy this pattern from the template and never choose the concrete types yourself—they are injected by the node-runner binary at startup.

3.6 Attribute macros

An **attribute macro** (`#[name]` or `#[name(args)]`) is placed above a type and runs at compile time to generate additional code, eliminating boilerplate.

Attribute	Effect
`#[ledgera_data]`	Registers the type as `LAT::Data`; derives serialisation and comparison traits.
`#[ledgera_computation]`	Registers the enum as `LAT::Compute`. Requires an `async fn ledgera_compute(&self, Vec<Data>) -> Result<Data, Error>`.
`#[ledgera_tag]`	Registers the enum as `LAT::Tag` (anchors inputs, no output).
`#[ledgera_local_predicate]`	Registers as `LAT::LocalPredicate`. Requires `fn ledgera_single_arg_is_valid(&self, &Data, ...) -> Result<bool, Error>`.
`#[ledgera_global_predicate]`	Registers as `LAT::GlobalPredicate`. Requires `fn ledgera_multi_args_is_valid(&self, &[&Data]) -> Result<bool, Error>`.
`#[ledgera_error]`	Registers the enum as `LAT::RuntimeError`; derives required traits.
`#[ledgera_application_template(...)]`	Generates the full `impl LedgeraApplicationTemplate` block (see §3.4), wiring all six associated types.

When you annotate your backend struct with `#[ledgera_application_template(...)]`, the macro generates an `impl` block, filling each slot with the concrete types you passed as arguments. Here we show the `imp` block that the macro generates for the `app_string_concat` example:

```
// Generated by #[ledgera_application_template(...)]
impl LedgeraApplicationTemplate for ToyStringComputationBackend {
    fn get_service_name(&self) -> &str { "string_concat" }
    type RuntimeError     = ToyStringRuntimeError;
    type Data             = ToyStringValue;
    type Compute          = ToyStringComputation;
    type Tag              = ToyStringTag;
    type LocalPredicate   = ToyStringLocalPredicate;
    type GlobalPredicate  = ToyStringGlobalPredicate;
}
```

All seven macros are applied together in `lat_binding.rs`; see §4.2 for a complete example.

3.7 Async Rust

Ledgera nodes run many concurrent tasks without blocking threads.

3.7.1 The async/await keywords

An `async fn` returns a **Future** driven by the `tokio` executor. The `.await` keyword suspends the current task until the future resolves, without blocking the OS thread:

```
pub async fn fetch_and_process() -> Result<String, Error> {
    let data = fetch_from_network().await;  // waits here
    Ok(process(data))
}
```

3.7.2 Multi-producer single-consumer (MPSC) channels

A **multiple-producer, single-consumer** (MPSC) channel sends values between tasks:

```
let (tx, mut rx) = tokio::sync::mpsc::channel::<u32>(32);
tx.send(42).await.unwrap();
if let Some(v) = rx.recv().await { println!("{}", v); }
```

In `ServiceClientRuntimeIO`, one channel carries user commands from the TUI into the service logic; another feeds results back the other way.

3.7.3 Exclusive access via `Arc<Mutex<T>>`

`Arc` is a thread-safe shared pointer; `Mutex<T>` enforces exclusive access. Together they allow mutable state to be shared across async tasks:

```
let shared = Arc::new(tokio::sync::Mutex::new(0u32));
let handle = shared.clone();
tokio::spawn(async move {
    *handle.lock().await += 1;  // exclusive write
    // lock released on drop
});
```

3.7.4 Method `tokio::select!`

Method `tokio::select!` waits on several async expressions at once and runs the body of whichever resolves first – the standard pattern for an event loop:

```
loop {
    tokio::select! {
        Some(msg) = rx_a.recv() => handle_a(msg).await,
        Some(msg) = rx_b.recv() => handle_b(msg).await,
    }
}
```

The service client uses exactly this pattern to multiplex peer messages, Core feedback, and user input (see §5.3).

4 Ledgera's Hello World application — `app_string_concat`

The application `app_string_concat` is the simplest possible Ledgera application: it concatenates two strings over a BFT network. Its purpose is to serve as the minimal reference for the `lat_binding.rs` file—the file that defines the domain-specific concepts the network is working with. Unlike the second example (`app_varkeep` from §5), `app_string_concat` has no custom client-side logic: high-level user requests directly correspond (one-to-one) to lower level requests on Ledgera.

In the remainder we show how the application is developped by following four steps, which include (1) specifying the application behaviour, (2) providing the binding as a Ledgera

Application Template, (3) defining the text-based user interface, and (4) defining and running the test harness.

4.1 Step 1 — Application design

1 Behavior — 2 Binding — 3 TUI — 4 Tests

> **Goal.** Answer three questions before touching any code: what data, what operations, what user actions?

Before writing any code, specify the behavior of your application by answering the following questions:

1. What **data** does the application manipulate?
2. What **operations** will the network run in a distributed, verifiable way?
3. What **user actions** will a client node expose?

For `app_string_concat`:

Question	Answer
Data	A single type `StrConcatData` wrapping a `String`.
Operations	One computation `Concat` (concatenate two strings into one), and one tag `Tag` (anchor a string without producing output).
User actions	The same two – `Concat` and `Tag` – driven from the TUI.

4.2 Step 2 — Implementing `lat_binding.rs`

1 Behavior — 2 Binding — 3 TUI — 4 Tests

> **Goal.** Turn the Step 1 design into code: the network vocabulary seen by the Ledgera Core, all in a single file.

The behavior of our example application being well-specified, we now introduce the binding file `lat_binding.rs` in the `app_string_concat` crate. The file is divided into six sections, each annotated with one of the Ledgera macros.

4.2.1 Data

One struct per application, wrapping the domain value. The macro generates all the serialisation and comparison traits that Core requires.

```
#[ledgera_data]
pub struct StrConcatData {
    pub string: String,
}
```

4.2.2 Errors

One enum listing every domain-specific error that a computation can produce. Add at least one catch-all variant.

```
#[ledgera_error]
pub enum StrConcatRuntimeError {
    AtomicConcatenationComputationRequiresExactlyTwoStrings,
    PairwiseDistinctConstraintEvaluationRequiresExactlyTwoStrings,
```

```
    PairwiseDistinctConstraintViolation,
}
```

4.2.3 Computations

One enum variant per computation. You must also provide an `async fn ledgera_compute` method whose signature is fixed: `Vec<Data>` → `Result<Data, RuntimeError>`. This is the code every Voter runs locally to replicate the execution.

```
#[ledgera_operation]
pub enum StrConcatComputation { Concat }

impl StrConcatComputation {
    pub async fn ledgera_compute(
        &self,
        arguments: Vec<StrConcatData>,
    ) -> Result<StrConcatData, StrConcatRuntimeError> {
        match self {
            StrConcatComputation::Concat => {
                if arguments.len() != 2 {
                    Err(
    StrConcatRuntimeError::
    AtomicConcatenationComputationRequiresExactlyTwoStrings
                    )
                } else {
                    let s1 = arguments[0].string.clone();
                    let s2 = arguments[1].string.clone();
                    Ok(StrConcatData {
                        string: format!("{}{}", s1, s2)
                    })
                }
            }
        }
    }
}
```

4.2.4 Tags

Tags take inputs but produce no computed output—they anchor data and produce a proof without running a computation.

```
#[ledgera_tag]
pub enum StrConcatTag { Tag }
```

4.2.5 Predicates

Fundamental rule

A predicate is a pure function: no I/O, no external state. It checks a constraint on one or more input values *before* the operation runs.

Two categories exist:

Macro	Role
#[ledgera_local_predicate]	Checked per argument. Method: `ledgera_single_arg_is_valid(&self, value: &Data, id: &Sig)`.
#[ledgera_global_predicate]	Checked across the whole input set. Method: `ledgera_multi_args_is_valid(&self, args: &[&Data])`.

For app_string_concat:

```
#[ledgera_local_predicate]
pub enum StrConcatLocalPredicate { StringLongerThan(u32) }

impl StrConcatLocalPredicate {
    pub fn ledgera_single_arg_is_valid(
        &self,
        value: &StrConcatData,
        _function_instance_identifier:
           &SerdeSerializable64BitsSignature,
    ) -> Result<bool, StrConcatRuntimeError> {
        match self {
            StrConcatLocalPredicate::StringLongerThan(min) =>
                Ok((value.string.len() as u32) >= *min),
        }
    }
}

#[ledgera_global_predicate]
pub enum StrConcatGlobalPredicate { PairwiseDistinct }

impl StrConcatGlobalPredicate {
    pub fn ledgera_multi_args_is_valid(
        &self,
        arguments: &[&StrConcatData],
    ) -> Result<bool, StrConcatRuntimeError> {
        if arguments.len() != 2 {
            Err(
    StrConcatRuntimeError::
    PairwiseDistinctConstraintEvaluationRequiresExactlyTwoStrings
            )
        } else {
            match self {
                StrConcatGlobalPredicate::PairwiseDistinct =>
                    Ok(arguments[0] != arguments[1]),
            }
        }
    }
}
```

4.2.6 Application template

The registry that ties all the above types together and generates the `impl LedgeraApplicationTemplate` block (see §3.4).

```
#[ledgera_application_template(
    name              = "string_concat",
    data              = StrConcatData,
    computation       = StrConcatComputation,
```

```
    tag               = StrConcatTag,
    local_predicate   = StrConcatLocalPredicate,
    global_predicate  = StrConcatGlobalPredicate,
    error             = StrConcatRuntimeError,
)]
#[derive(Debug)]
pub struct StrConcatBackend {}
```

4.3 Step 3 — text-based user interface (TUI)

1 Behavior — 2 Binding — **3 TUI** — 4 Tests

For the `app_tring_concat` application, high-level user requests correspond one-to-one to low level requests to Ledgera. As a result, we only use a very simple TUI, that specializes generic code written in the `util_basic_tui` crate. This specialization corresponds to the `app_tring_concat_tui` crate which only contains two files:

- `parser.rs` in which we write code that can convert text entered by the user to concat or tag commands
- `printer.rs` in which we indicate how to represent domain-specific objects (StrConcatData, StrConcatComputation etc) in the TUI

4.4 Step 4 — Running the test harness

1 Behavior — 2 Binding — 3 TUI — **4 Tests**

> **Goal.** Spin up a local 4-node BFT network and interact with the application through the TUI.

Once the behaviour is specified and implemented by providing a service binding, we must provide a test harness. Such a test harness lives in `app_string_concat_test/` and contains two binaries and a launch script. The same structure is reused in every subsequent example (§5.5) and in the blank template (§6.6); only the topology and binary names change.

4.4.1 Network topology

Each node plays one or more roles.

Node	Roles
`node1`	Client + Voter
`node2`	Storer + Voter
`node3`	Client + Storer + Voter
`node4`	Voter + Logger

4.4.2 Binaries and launch

From `crates/app_string_concat_test/`, we can lauch the test with the command:

```
python launch_test.py
```

Internally, this Python script generates the cryptographic keys and launches the nodes, with the following commands:

- `testnet_maker 4`—generates PKI key material for 4 nodes under `./testnet/`.
- `configurable_node_runner <name> <roles> <pki_folder>`—launches one node.

The script builds the binaries if needed (`cargo build --release`), calls `testnet_maker`, and opens one terminal per node via `gnome-terminal` (Linux) or `start cmd` (Windows).

5 An Atomic Register over Ledgera — app_varkeep

The example application `app_varkeep` implements a distributed variable store: clients can assign key-value pairs either *locally* (just publishing to peers, no BFT agreement) or *globally* (going through Core, so every honest node eventually agrees on the assignment).

5.1 Step 1 — Application design

1 Behavior — 2 Binding — 3 TUI — 4 Tests

> **Goal.** Decide on data, operations, and user actions before writing any code.

Question	Answer
Data	An enum `VarkeepData` with two variants: `VariableName(String)` and `VariableValue(String)`.
Operations	No real computation (a `Placeholder` operation that always errors). One tag `Assign`: atomically anchors a `(name, value)` pair via Core.
User actions	`AssignLocal(name, value)` – publish directly to other Varkeep clients, no BFT agreement. `AssignGlobal(name, value)` – submit `TagInputs(Assign)` to Core; all honest nodes eventually agree.

The application prescribes two assignment operations, with different *local* and *global* modes: a local assignment is cheap and immediate but does not involve a shared variable; a global assignment is more expensive but can write on a shared variable and produces a Ledgera Proof that is agreed upon by a BFT quorum.

5.2 Step 2a — Implementing lat_binding.rs

1 Behavior — **2 Binding** — 3 TUI — 4 Tests

> **Goal.** Define the network vocabulary: data, errors, operations, tags, predicates.

This step follows the lines of the same step for application `app_string_concat` (§ 4.2). Only the types and their semantics differ.

```
#[ledgera_data]
pub enum VarkeepData {
    VariableName(String),
    VariableValue(String),
}

#[ledgera_error]
pub enum VarkeepRuntimeError { DefaultError }

#[ledgera_computation]
pub enum VarkeepComputation { Placeholder }

impl VarkeepComputation {
    pub async fn ledgera_compute(
        &self, _arguments: Vec<VarkeepData>,
```

```
    ) -> Result<VarkeepData, VarkeepRuntimeError> {
        Err(VarkeepRuntimeError::DefaultError)
    }
}

// Anchors a (VariableName, VariableValue) pair
#[ledgera_tag]
pub enum VarkeepTag { Assign }

// Check that each input is the right variant
#[ledgera_local_predicate]
pub enum VarkeepLocalPredicate {
    IsVarName, IsVarValue
}

impl VarkeepLocalPredicate {
    pub fn ledgera_single_arg_is_valid(
        &self, value: &VarkeepData,
        _id: &SerdeSerializable64BitsSignature,
    ) -> Result<bool, VarkeepRuntimeError> {
        Ok(match self {
            Self::IsVarName  => matches!(value,
               VarkeepData::VariableName(_)),
            Self::IsVarValue => matches!(value,
               VarkeepData::VariableValue(_)),
        })
    }
}

// No cross-argument constraint needed.
#[ledgera_global_predicate]
pub struct VarkeepGlobalPredicate;

impl VarkeepGlobalPredicate {
    pub fn ledgera_multi_args_is_valid(
        &self, _arguments: &[&VarkeepData],
    ) -> Result<bool, VarkeepRuntimeError> { Ok(true) }
}

#[ledgera_application_template(
    name             = "varkeep",
    data             = VarkeepData,
    computation      = VarkeepComputation,
    tag              = VarkeepTag,
    local_predicate  = VarkeepLocalPredicate,
    global_predicate = VarkeepGlobalPredicate,
    error            = VarkeepRuntimeError,
)]
#[derive(Debug, Clone)]
pub struct LedgeraVarkeepService;
```

5.3 Step 2b — Implementing `service_client_logic/`

1 Behavior — 2 Binding — 3 TUI — 4 Tests

Goal. Implement the client-side logic: the state machine that connects user input, Core feedback, and peer messages.

This module is entirely absent from app_string_concat, which has a trivial client logic (the client operations directly invoke the corresponding primitives of Core) and therefore lacks a service_client_logic folder; the client logic is the heart of app_varkeep, which is structured as follows:

```
src/service_client_logic/
├── mod.rs
├── role.rs - one string constant naming this role
├── user_reqs.rs - high-level user request enum
├── runtime_io.rs - IO handle returned to the TUI
├── service_msgs/ - peer message types and Zenoh topics
├── behavior/ - the three react_to_* handlers
└── service_client_state.rs - event loop
```

5.3.1 Files user_reqs.rs and runtime_io.rs

File user_reqs.rs declares the high-level user request enum—one variant per action a user can trigger from the TUI:

```
pub enum HighLevelVarkeepUserRequests {
    AssignLocal(String, String),   // peer-to-peer only
    AssignGlobal(String, String),  // through Core (BFT)
}
```

File runtime_io.rs defines the handle returned to the TUI after the service client starts. It exposes two MPSC endpoints: one to send user requests in the "in" direction, and one to receive (key, value) display updates in the "out" direction:

```
pub struct ServiceClientRuntimeIO {
    pub user_requests_sender: Sender<HighLevelVarkeepUserRequests>,
    pub tui_feed_receiver:    Receiver<(String, String)>,
}
```

5.3.2 Client messages — Folder service_msgs/

File messages.rs in service_msgs/ defines the messages that clients exchange with each other directly (outside the control of the Voters). Each type implements LedgeraPublishableMessage with a unique type string:

```
pub struct LedgeraVarkeepServicePublishLocVarMsg {
    pub varname: String, pub varvalue: String,
}
impl LedgeraPublishableMessage for
   LedgeraVarkeepServicePublishLocVarMsg {
    fn get_msg_type() -> &'static str { "servicePublishPos" }
}
```

For peer to peer communications, we use a Publish Subscribe pattern and thus, we need to define topics. The file topics.rs in he same folder declares these topic strings:

```
pub enum VarkeepServicesDedicatedTopics { PublishLocalVariable }

impl VarkeepServicesDedicatedTopics {
    pub fn get_topic_str(&self, _name_as_client: &str) -> String {
```

```
        match self {
            Self::PublishLocalVariable =>
                "varkeep/publishLocVar".to_string(),
        }
    }
}
```

5.3.3 The Client behavior — Folder `behavior/`

The struct `LedgeraServiceClientBehavior` in file `behavior/mod.rs` holds all runtime handlers the three `react_to_*` methods share: the Publish-Subsctibe communication session, the PKI parameters, the service descriptor, the `CoreClientRuntime`, and the TUI feed sender.

A helper `submit_and_log` method of the same struct assembles a `LedgeraAtomicOperationSpecification` from an operation and its argument list, calls `core_client_runtime_io.compute_function(...)`, and emits a log line. All branches in the pattern-mathcing in `react_to_user` that invoke Core primitives call this helper.

5.3.4 Dispatching requests — File `react_to_user.rs`

The implementation of struct `LedgeraServiceClientBehavior` in this file dispatches each `HighLevelVarkeepUserRequests` variant to its network action:

```
match service_user_req {
    // Local: publish directly to peers
    AssignLocal(varname, varvalue) => {
        let msg = LedgeraVarkeepServicePublishLocVarMsg::new(
            varname,
            varvalue
        );
        self.comm_session.lock().await
        .serialize_and_publish_on_topic::<
            LedgeraVarkeepServicePublishLocVarMsg
        >(
            &self.comm_params,
            &VarkeepServicesDedicatedTopics::PublishLocalVariable
            .get_topic_str("NA"),
            &msg,
        ).await;
    }
    // Global: build a TagInputs(Assign) spec and submit to Core.
    AssignGlobal(varname, varvalue) => {
        let spec = LedgeraAtomicOperationSpecification::new(
            LedgeraAtomicOperation::TagInputs(
                LedgeraVarkeepServiceTag::Assign
            ),
            None,
            vec![
        LedgeraInputArgument::RawValue {
            is_input_persistent: false,
            value: VarkeepData::VariableName(varname),
        },
        LedgeraInputArgument::RawValue {
            is_input_persistent: false,
            value: VarkeepData::VariableValue(varvalue),
        },
```

```
            ],
        );
        let _ =
            self.core_client_runtime_io.compute_function(spec).await;
    }
}
```

5.3.5 Dispatching Core messages — File react_to_core.rs

The implementation of struct `LedgeraServiceClientBehavior` in this file dispatches each `ValidatedCoreFeedbackMessage` that arrives from Core. For a global assignment, the relevant variant is `ValidatedComputationInstance`; this variant carries the original spec (and therefore the agreed-upon name and value), which are forwarded to the TUI via the method `to_ui_feed`:

```
match validated_core_msg {
    ValidatedCoreFeedbackMessage::ValidatedComputationInstance(inst)
       => {
        let varname  = inst.rfun.spec.arguments.first().unwrap();
        let varvalue = inst.rfun.spec.arguments.get(1).unwrap();
        if let (RawValue { value: VariableName(n), .. },
                RawValue { value: VariableValue(v), .. }) =
                   (varname, varvalue) {
            let _ = self.to_ui_feed
                .send((format!("core.{}", n),
                   v.to_string())).await;
        }
    }
    _ => {}
}
```

The `"core."` prefix distinguishes globally-agreed values from locally-published ones (with a `"clientN."` prefix) in the TUI display.

5.3.6 Dispatching peer messages — File react_to_peer.rs

The implementation of struct `LedgeraServiceClientBehavior` in this file dispatches each message received by a peer (Client-to-Client communication). The sender's public key is resolved to a stable client index that will be displayed bu the TUI:

```
let key = format!("client{}.{}", client_id, publish_msg.varname);
let _ = self.to_ui_feed.send((key, publish_msg.varvalue)).await;
```

5.3.7 The event loop — File service_client_state.rs

`LedgeraServiceClientState::run()` subscribes to the relevant Zenoh topics, spawns the main event loop, and returns `ServiceClientRuntimeIO` to the caller. The loop is a `tokio::select!` over three sources, which correspond to the three handlers `react_to_*` from above :

```
loop {
    tokio::select! {
        Some((msg, sig)) = peer_receiver.recv() => {
            behavior.react_to_service_msg_publish_local_var(msg,
               sig).await;
        }
        Some(core_msg) = core_receiver.recv() => {
            behavior.react_to_validated_core_msg(core_msg).await;
```

```
        }
        Some(user_req) = user_receiver.recv() => {
            behavior.react_to_service_user_req(user_req).await;
        }
    }
}
```

5.4 Step 3 — The text-based user interface (TUI)

1 Behavior — 2 Binding — **3 TUI** — 4 Tests

> **Goal.** Give client nodes a terminal interface that parses typed commands and renders the variable map.

The text-based user interface (TUI)—which is based on the Rust library Ratatui—lives in `app_varkeep_tui/`. The rendering and lifecycle code is standard Ratatui boilerplate; only the three files below contain domain-specific logic.

5.4.1 Commands — Folder `commands/`

File `commands/tui_commands.rs` contains one enum variant per user action, mirroring `HighLevelVarkeepUserRequests` in `app_varkeep`:

```
pub enum LedgeraVarkeepServiceTuiCommand {
    Exit,
    AssignLocal(String, String),
    AssignGlobal(String, String),
}
```

Our parser is based on the Rust library `nom`. The parser maps typed text to variants. In file `commands/parse_command.rs`, each command keyword is followed by two whitespace-separated alphanumeric tokens:

```
fn parse_inner<'a, E: ParseError<&'a str>>(input: &'a str)
    -> IResult<&'a str, LedgeraVarkeepServiceTuiCommand, E>
{
    alt((
        map(tag("exit"), |_|
            LedgeraVarkeepServiceTuiCommand::Exit),
        map(
            (preceded(tag("locassign"), preceded(multispace1,
                alphanumeric1)),
             preceded(multispace1, alphanumeric1)),
            |(vn, vv)| AssignLocal(vn.to_string(), vv.to_string()),
        ),
        map(  // same structure for "gloassign" -> AssignGlobal
            (preceded(tag("gloassign"), preceded(multispace1,
                alphanumeric1)),
             preceded(multispace1, alphanumeric1)),
            |(vn, vv)| AssignGlobal(vn.to_string(),
                vv.to_string()),
        ),
    )).parse(input)
}
```

5.4.2 File app.rs

The method `process_command()` translates each parsed `TuiCommand` into a `HighLevelVarkeepUserRequests` sent over `user_requests_sender`:

```
match cmd {
    AssignLocal(vn, vv) =>
        sender.send(HighLevelVarkeepUserRequests::AssignLocal(vn,
            vv)).await,
    AssignGlobal(vn, vv) =>
        sender.send(HighLevelVarkeepUserRequests::AssignGlobal(vn,
            vv)).await,
    Exit => { /* ... */ }
}
```

The method `listen_and_update()` in the same file uses `tokio::select!` to handle both terminal key events and feedback arriving from the service client:

```
tokio::select! {
    event = self.event_stream.next().fuse() => { /* handle
        keyboard input */ }
    Some((vn, vv)) =
        self.service_client_runtime_io.tui_feed_receiver.recv() => {
         self.varmap.insert(vn, vv);  // update the displayed
            variable map
    }
    _ = tokio::time::sleep(Duration::from_millis(100)) => {}
}
```

5.4.3 File setup.rs

File `setup.rs` provides two thin wrappers called by the node runner: one to initialise the logger before the node starts, and one to launch the Ratatui terminal once `run()` has returned a `ServiceClientRuntimeIO`.

5.5 Step 4 — Running the test harness

1 Behavior — 2 Binding — 3 TUI — **4 Tests**

> **Goal.** Spin up a local 4-node network and exercise the AssignLocal and AssignGlobal commands.

This step follows the lines of Step 3 in §4.4; only the topology and binary names differ. The test harness lives in `app_varkeep_test/`.

Node	Roles
`node1`	Client + Voter (`cv`)
`node2`	Client + Storer + Voter (`csv`)
`node3`	Client + Storer (`cs`)
`node4`	Voter + Logger (`vl`)

Use commands `service_varkeep_testnet_maker 4 3` and `service_varkeep_node_runner` to generate the binaries, and run `python launch_test.py` from `crates/app_varkeep_test/` to launch the tests.

Once the four terminals are open, try the following operations from any client node:

- `locassign myvar hello`—immediately visible to peers as `clientN.myvar = hello`.
- `gloassign myvar world`—after BFT agreement via Core, appears as `core.myvar = world` on all nodes.

6 Building your own application

The three crates `blank_app_template`, `blank_app_template_tui`, and `blank_app_template_test` form a skeleton that compiles out of the box and mirrors the exact structure you explored in Sections 4 and 5. You can follow the four steps above and develop your own application. Every file contains `TODO` comments that mark precisely what to replace.

6.1 Setting up the crates

> **Goal.** Obtain three compiling, workspace-registered crates by copying the templates and renaming them – no domain logic yet, just the right skeleton.

Before delving into the four steps described above, make sure you obtain three workspace-registered, compiling crates by copying the templates and renaming them with the name of your application—no application logic yet, just the right skeleton.

6.1.1 Copy the three template crates

```
crates/blank_app_template/ → crates/app_myapp/
crates/blank_app_template_tui/ → crates/app_myapp_tui/
crates/blank_app_template_test/ → crates/app_myapp_test/
```

6.1.2 Update each `Cargo.toml`

In `crates/app_myapp/Cargo.toml`:

```
[package]
name    = "ledgera-app-myapp"
edition = "2021"

[dependencies]
ledgera-types       = { workspace = true }
ledgera-core-logic  = { workspace = true }
ledgera-node-client = { workspace = true }
ledgera-comms       = { workspace = true }
ledgera-pki         = { workspace = true }
ledgera-macros      = { workspace = true }
nom                 = { workspace = true }
serde               = { workspace = true }
rand                = { workspace = true }
tokio               = { workspace = true }
bincode             = { workspace = true }
log                 = { workspace = true }
hex                 = { workspace = true }
```

In `crates/app_myapp_tui/Cargo.toml`, update the name and the dependency that points to the application crate:

```
[package]
name = "ledgera-app-myapp-tui"
[dependencies]
# ... (keep all other dependencies unchanged)
```

```
ledgera-app-myapp = { workspace = true }  # was:
   ledgera-blank-app-template
```

In `crates/app_myapp_test/Cargo.toml`:

```
[package]
name = "ledgera-test-app-myapp"
[dependencies]
# ... (keep all other dependencies unchanged)
ledgera-app-myapp     = { workspace = true }
ledgera-app-myapp-tui = { workspace = true }
```

6.1.3 Register in the workspace

In the root `Cargo.toml`:

```
# [workspace] members:
"crates/app_myapp",
"crates/app_myapp_tui",
"crates/app_myapp_test",

# [workspace.dependencies]:
ledgera-app-myapp     = { package = "ledgera-app-myapp",
                          path    = "crates/app_myapp" }
ledgera-app-myapp-tui = { package = "ledgera-app-myapp-tui",
                          path    = "crates/app_myapp_tui" }
```

Then verify: `cargo check` from the workspace root.

6.2 Step 1 — Application design

1 Behavior — 2 Binding — 3 TUI — 4 Tests

> **Goal.** Answer the same three questions before writing any domain logic.

Question	app_string_concat	app_varkeep	Your app
Data	`StrConcatData`	`VarkeepData`	
Operations	`Concat` + `Tag`	placeholder + `Assign` tag	
User actions	generic TUI	`AssignLocal`, `AssignGlobal`	

6.3 Step 2a — Implementing `lat_binding.rs`

0 Crate — 1 Behavior — 2 Binding — 3 TUI — 4 Tests

> **Goal.** Fill in every `TODO` in `src/lat_binding.rs`. This is the only mandatory file; skip Step 2b if you need no custom client logic.

Item	What to do	See
`#[ledgera_data]`	Replace `Placeholder` with your data variants (struct or enum).	*§4 Step 2*
`#[ledgera_error]`	Add one variant per failure mode; keep a catch-all.	*§4 Step 2*
`#[ledgera_operation]` + `ledgera_compute`	One variant per computation; implement the `match` body. Signature is fixed: `Vec<Data>` → `Result<Data,Error>`.	*§4 Step 2*
`#[ledgera_atomic_tag]`	One variant per `TagInputs` operation; omit if unused.	*§5 Step 2a*
`#[ledgera_local_predicate]`	Per-argument constraints; return `Ok(true)` if unused.	*§4 Step 2*
`#[ledgera_global_predicate]`	Cross-argument constraints; return `Ok(true)` if unused.	*§4 Step 2*
`#[ledgera_application_template]`	Set a unique `name` string; wire all six types.	*§4 Step 2*

6.4 Step 2b — Implementing `service_client_logic/`

0 Crate — 1 Behavior — **2 Binding** — 3 TUI — 4 Tests

> **Goal.** Implement the client-side state machine, following the `app_varkeep` pattern (§5 Step 2b).

File	What to do
`user_reqs.rs`	Add one `HighLevelServiceUserRequests` variant per user action.
`service_msgs/messages.rs`	Add fields to `LedgeraServiceTemplateType1Message`, or add further structs annotated with `#[ledgera_service_msg]`.
`service_msgs/topics.rs`	Declare one Zenoh topic constant per peer message type.
`behavior/mod.rs`	Add internal-state fields to `LedgeraServiceClientBehavior`; use `submit_and_log` to reach Core.
`behavior/react_to_user.r`	Match on each user request: call `submit_and_log` for Core ops, or publish a peer message for local-only actions.
`behavior/react_to_core.r`	Match on `ValidatedCoreFeedbackMessage` variants; forward results to the TUI feed.
`behavior/react_to_peer.r`	Match on incoming peer messages; update state and/or forward to TUI.

6.5 Step 3 — The text-based user interface (TUI)

0 Crate — 1 Behavior — 2 Binding — **3 TUI** — 4 Tests

> **Goal.** Wire user-typed commands to `HighLevelServiceUserRequests` variants, following the `app_varkeep_tui` pattern (§5 Step 3).

- `commands/tui_commands.rs` – add one enum variant per command, carrying its arguments.

- `commands/parse_command.rs` – add one `map(...)` arm inside `alt((...))` for each new variant.

6.6 Step 4 — Running the test harness

0 Crate — 1 Behavior — 2 Binding — 3 TUI — **4 Tests**

> **Goal.** Adapt `launch_test.py` to your topology and run the testnet.

Two changes are needed before running:

1. Rename the binary constants at the top of `launch_test.py` (`TESTNET_MAKER`, `NODE_RUNNER`) to match the binary names in your `Cargo.toml`.
2. Edit the `NODES` list to match your desired topology (see §4.4 for the role letter codes).

Then, from `crates/app_myapp_test/`:

```
python launch_test.py
```

Checklist before launching

- ☐ `lat_binding.rs`: all `TODO`s replaced; `name` in `#[ledgera_application_template]` is unique.
- ☐ `ledgera_compute`: every variant returns `Ok(...)`, not `Err(DefaultError)`.
- ☐ `react_to_user`: every `HighLevelServiceUserRequests` variant is handled.
- ☐ Parser covers every `tui_commands` variant.
- ☐ `TESTNET_MAKER` / `NODE_RUNNER` match binary names in `Cargo.toml`.
- ☐ `cargo build --release` succeeds from the workspace root.