BEST PRACTICE
EVENT-DRIVEN ARCHITECTURE FOR ENTERPRISE

Communication between services in a microservice architecture has an instrumental role. Inefficient, too tightly coupled or even too loosely coupled communication results in a system, which is difficult and costly in development and maintenance.

INTRODUCTION

In an event-driven architecture (EDA) components communicate publishing and subscribing to events rather than through tightly coupled calls. This approach naturally decouples services, enhances scalability, and improves responsiveness. By allowing asynchronous reactions to events, an EDA supports more real-time processing of data and enables independent components to work cohesively.

The following sections describe how Noumena‘s event-driven architecture supports event-driven architecture key principles – decentralized services, task-based completion, parallel processing with data-driven dependencies.

KEY PRINCIPALS

Loosely coupled

The Noumena Engine is integrating with the external systems with AMQP messages, which are specified in the business logic described in NPL protocols

Process exactly once

A critical feature of any financial system is preventing the event to be lost unnoticed and, in many cases much more importantly, ensuring the processing of the event is done only once

Service independence

Services are listening on relevant events (messages) and act accordingly on them, that is, interacting with external systems like blockchains, banking systems, etc. Those services are Integration services

Decentralized services

Each service operates independently, reacting to events rather than relying on synchronous calls to other services. This allows teams to develop, deploy, and scale services independently

EVENT BASED BUSINESS PROCESS LIFECYCLE

The lifecycle events triggered by a protocol are divided into two categories:

1. Protocol state changes
If the protocol state is changed, a Server Sent Event (SSE) is emitted. The consumer may subscribe for those events and use them to update the UI or notify external systems.

2. User defined notifications
NPL allows users to trigger custom notifications that are being propagated by AMQP message broker. In order to trigger a notification it needs to be declared in the scope

```
notification PayInterests(iban: Text, amount: Number) returns Unit;
```

The parameters for notification declaration are specified depending on the needs of the event consumer. The notification can be then triggered by invoking:

```
notify PayInterests(iban, interests);
```

The NOUMENA Runtime will publish an AMQP message to the broker only when the requested action is successfully completed and committed to the database and hence guarantee transaction safety.

Very often just triggering an event is not enough and followup actions need to take place. For such requirements notifications can bear the callback information:

```
notification PayInterests(iban: Text, amount: Number)
return TypeOfCallbackArgument;
```

And the trigger with **`notify:`**

```
notify PayInterests(iban, interests) resume userAction;
```

The **`userAction`** is an action (permission) defined in the protocol sending the notification.

LIFECYCLE EVENTS TRIGGERED BY EXTERNAL SYSTEMS

Reacting to external events, i.e., receiving information from external systems, is accomplished through REST calls. Because the NOUMENA Runtime guarantees that no blocking I/O operations occur during transactional processing, REST requests are often more advantageous than fully decoupled messaging systems. This is particularly true when short-lived JWTs are used. When an external event is detected by the integration service, it triggers the corresponding action by sending a REST request to the NOUMENA Runtime. This request must include a valid JWT token containing claims relevant to the involved party.

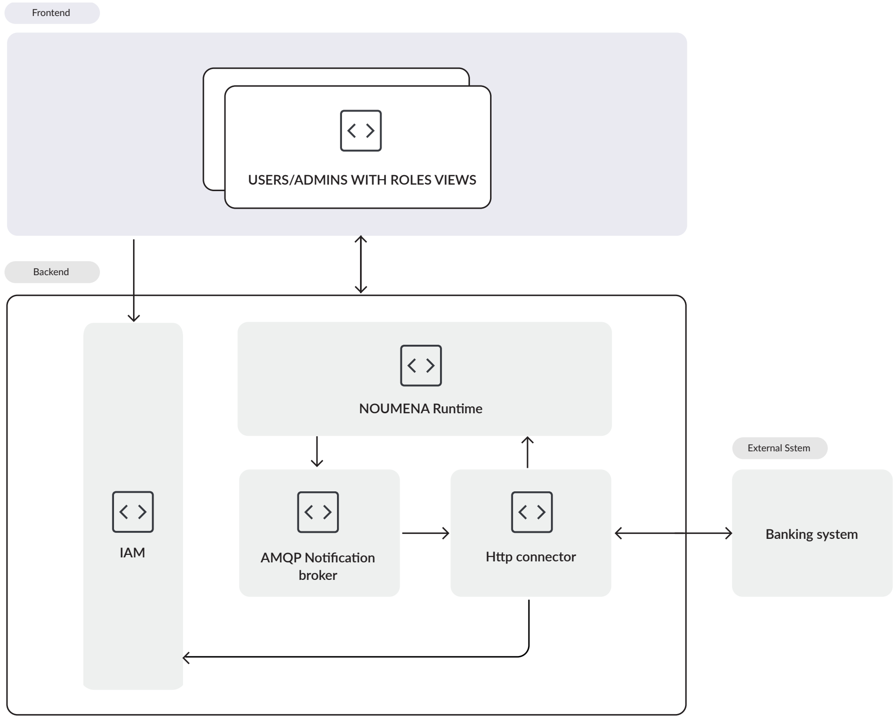
INTEGRATING EXTERNAL SERVICES

Architecture
External services are integrated with the NOUMENA Runtime through **Connectors**, a family of backend services designed for specific or generic integration scenarios.

For generic integrations with HTTP-based external services (e.g., REST APIs), the **HTTP Connector** is the preferred approach.

The HTTP Connector provides a **bidirectional interface** for interacting with API providers. It listens for AMQP messages (i.e. NPL notifications) and calls back to the NOUMENA Runtime by invoking NPL permissions or obligations.

Upon receiving a relevant NPL notification, the HTTP Connector performs the corresponding API call to the external system. When a response is received, the connector provides feedback to the NOUMENA Runtime by invoking the NPL notification callback. The component diagram below presents a high-level overview of this architecture:



Depending on the requirements, once a message is received and/or processed by an integration service, the service invokes a callback to the NOUMENA Runtime. The NOUMENA Runtime then updates the state of the process using an atomic write operation. If the integration service fails (e.g., due to a crash), the event can be safely retried as part of the business process. The persisted state of the business or low-level process indicates whether the event has already been processed.

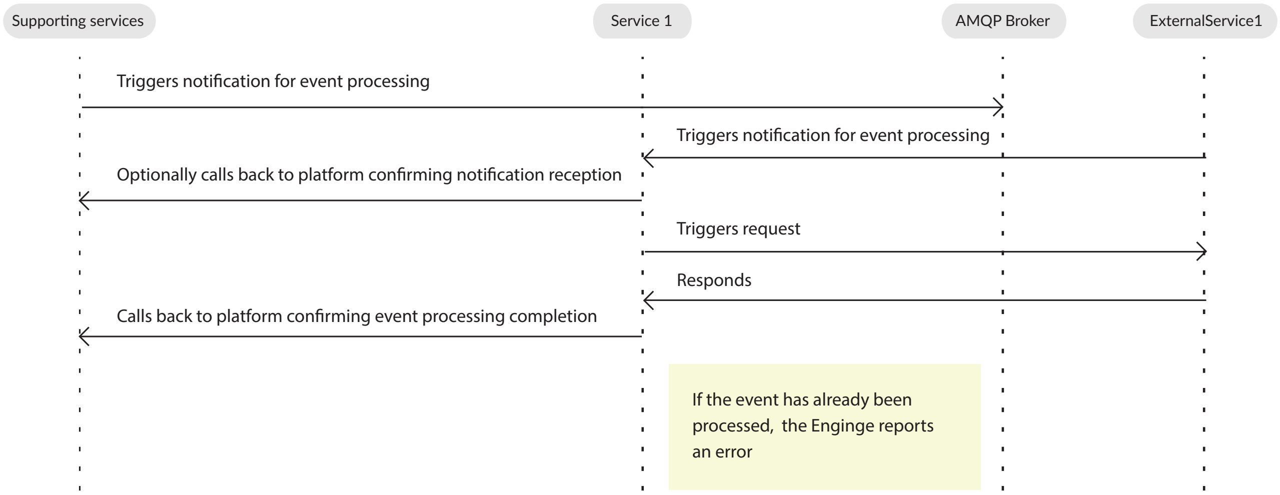
In addition to the HTTP Connector, other integration services are available, such as the **Blockchain Connector** and **Wallet Connector**.

EXACTLY ONCE PROCESSING OF EXTERNAL SERVICE CALLS

In any event-driven system, **exactly-once processing** is fundamental to ensuring robustness and operational integrity.

For many applications, e.g. such as financial systems, processing an event exactly once is more critical than the risk of losing the event entirely.

The Noumena Runtime and its ecosystem guarantee exactly-once processing through transactional state management and controlled callback mechanisms. The sequence diagram below illustrates one possible implementation of this approach:


Supporting services
Service 1
AMQP Broker
ExternalService1
Triggers notification for event processing
Triggers notification for event processing
Optionally calls back to platform confirming notification reception
Triggers request
Responds
Calls back to platform confirming event processing completion
If the event has already been processed, the Enginge reports an error

ERROR HANDLING

Error handling is a broad and complex topic, often one of the most overlooked and underestimated aspects of modern software systems. Designing a system that reacts appropriately to different types of errors and communicates them clearly to the right audience is particularly challenging due to the multiple levels and dimensions involved.

Below is a simplified overview of the main categories of errors:

1. Internal system and infrastructure errors
These errors are typically not exposed to end users. Instead, they should be caught and handled internally.
The NOUMENA Runtime and its ecosystem provide full observability, including metrics, alerting, and diagnostics to support resolution and monitoring.

2. User errors
Users may perform unexpected or invalid actions, including those not anticipated during system design.
To mitigate this, the NOUMENA Runtime includes robust safeguards: `require` conditions, which are assertions that verify boolean expressions at NOUMENA Runtime. If a condition fails, a custom error message is generated and returned as a REST response. These errors must be handled appropriately by the backend integration services or frontend

SCHEDULING

Many business processes require scheduling, for example, interest payouts, bill generation, or deadline enforcement.

The Noumena Runtime includes an internal transactional task scheduler that manages the execution of obligations.

Obligations, along with permissions, form the foundation of deontic logic within NPL (Noumena Protocol Language). In NPL, obligations are defined as methods that can be invoked by authorized parties. For example:

```
@api
obligation[operator] confirmCreditCheck(documentsLink: Text) before endDate |
requested {
    // Business logic
    become verified;
} otherwise become rejected;
```

In this case, if the `confirmCreditCheck` obligation is not invoked before the `endDate`, the Engine will automatically execute the `otherwise` block, transitioning the protocol instance to the `rejected` state.

Integration services can respond to such state transitions in two main ways:
1. Server-Sent Events (SSE): Listen for real-time state changes emitted by the Engine.
2. Polling: Periodically query protocol instances or types via a backend service to detect state changes.

The preferred approach (SSE vs. polling) depends on specific application requirements, such as latency, scalability, and infrastructure constraints.
Once a state change is observed, the backend service may react by invoking a corresponding **permission** to trigger the next business action.

This mechanism can also support **periodic tasks**, such as scheduled interest payments. To implement this, state transitions must be explicitly modeled. For example, in a periodic payment flow, the protocol may include states such as `waiting_for_payment` and `ready_for_payment.`

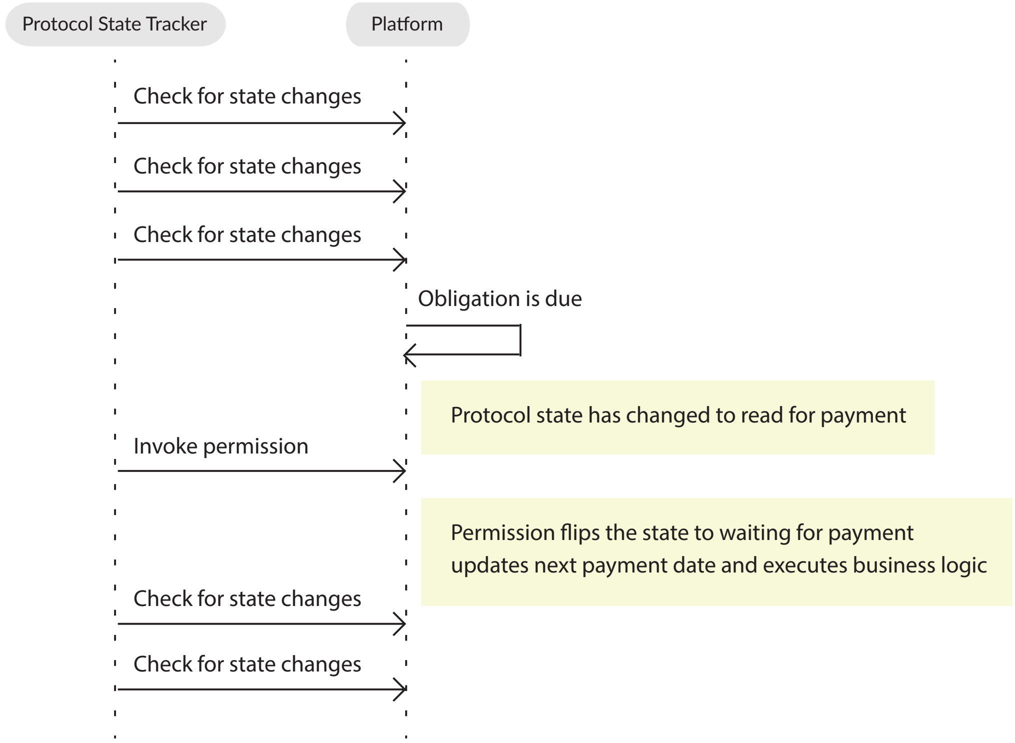
An obligation can be defined to trigger at the payout deadline, automatically transitioning the state to ready_for_payment, as shown below:

```
obligation[issuer] allowPayingInterests() before nextPayoutDate | waiting_
for_payment {
    } otherwise become ready_for_payment
```

A **Protocol State Tracker** monitors protocol instances for state transitions. When the state changes to `ready_for_payment`, the tracker invokes a corresponding **permission** to execute the actual payout logic. This permission should:

- Execute business logic (e.g., distribute interests),Update the `nextPayoutDate`
- Update the nextPayoutDate,
- Transition the protocol back to the `waiting_for_payment` state (directly or via chained permission calls)

The process is illustrated on the right:



This cyclical pattern enables recurring payments with full control and traceability.

The system is resilient by design, as the invocation logic is inherently **idempotent**: even if the same permission is triggered multiple times, only the first valid execution will proceed, while all others will be rejected based on the current protocol state. This protects against race conditions and ensures that external systems can retry safely without causing inconsistent outcomes.

If a crash occurs or multiple state trackers attempt to invoke the permission concurrently, only one invocation will succeed. Any subsequent calls will fail gracefully—returning an error indicating the protocol is no longer in the expected state, without side effects. This behavior is guaranteed by the NOUMENA Runtime internal transactional task scheduler, which ensures safe, atomic execution and prevents duplicated effects.

As a result, system designers only need to ensure that the Protocol State Tracker remains operational (e.g., via standard monitoring and alerting). The underlying scheduling, queuing, and execution guarantees are fully managed by the NOUMENA Runtime.

Get the KEY PRINCIPALS on the next page.

KEY TAKEAWAYS

The Noumena Runtime Event-driven-architecture enables scalable, reactive systems

By decoupling services through asynchronous event flows, the platform can process decoupled tasks in parallel due to optimistic locking, respond in real time, and scale independently. This model reduces tight inter-service dependencies, supports modular growth, and provides clear, auditable traces of system behavior, which is ideal for complex financial workflows where flexibility and reliability are essential.

The NOUMENA Runtime guarantees transactional safety and exactly-once semantics

Through atomic state transitions and commit-guarded notifications, the Engine ensures that events are only published when associated business logic has been successfully executed. This eliminates duplication and inconsistency that is critical in many systems.

User-defined notificationssimplify integration

System designers can expose high-level integration points by defining typed notifications directly in NPL. This makes external integrations (e.g., payments, document signing) easier to model, monitor, and extend.

The platform bridges system and human actions seamlessly

NPL supports defining obligations that can be scheduled or fulfilled by users. If unmet, the Engine applies fallback protocol state change. This design unifies technical workflows and human decision points under the same event lifecycle.

Built-in observability and error handling improve resilience

The platform provides structured error categories, runtime assertions (require), and full tracing across internal and external calls. This makes failures transparent and actionable across teams and users.

Scheduling and protocol state transitions are deterministic and audit-friendly

Time-based obligations ensure business events (e.g., credit checks, interest payouts) occur predictably.
All state changes are traceable, enabling auditability required in regulated environments.