NDH 事件模型與 TSDB 選型設計更新

版本: 1.1
日期: 2025-10-22
作者: Michael Lin 林志錚
組織: HTFA/Digital Twins Alliance

---

1. 概述

本文件針對 NDH（Neutral Data Hub）的事件模型與 TSDB（時序數據庫）選型進行詳細設計，解決原始設計中的以下問題：

1.1 原始設計的問題

In-Memory Event Bus + SQLite 的限制： - ❌ 事件 schema 未標準化 - ❌ 缺少事件重送機制 - ❌ 沒有事件持久化 - ❌ 不支援事件重放 - ❌ SQLite 作為時序查詢不夠優化

1.2 改進方案

本設計提供以下改進：

✅ **事件契約（IDL）**：使用 JSON Schema 定義所有事件類型
✅ **抽象 Event Bus 介面**：支援多種實作（In-Memory, ZMQ, MQTT）
✅ **TSDB 抽象與演進路徑**：SQLite → DuckDB/Parquet → TDEngine/InfluxDB/AVEVA PI
✅ **事件持久化與重放**：支援事件日誌和重放機制
✅ **事件重送與可靠性**：支援 at-least-once 和 exactly-once 語義

---

2. 事件契約（Event IDL）設計

2.1 事件基礎結構

所有事件都遵循統一的基礎結構，使用 JSON Schema 定義：

2.1.1 基礎事件 Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://idtf.org/schemas/events/base-event.json",
  "title": "Base Event",
  "description": "所有 IDTF 事件的基礎結構",
  "type": "object",
  "required": [
    "event_id",
    "event_type",
    "timestamp",
    "source",
    "version"
  ],
  "properties": {
    "event_id": {
      "type": "string",
      "format": "uuid",
      "description": "事件的唯一識別碼（UUID）"
    },
    "event_type": {
      "type": "string",
      "description": "事件類型（例如：TagValueChanged, InstanceCreated）"
    },
    "timestamp": {
      "type": "string",
      "format": "date-time",
      "description": "事件發生的時間戳（ISO 8601 格式）"
    },
    "source": {
      "type": "string",
      "description": "事件來源（例如：NDH, IADL_Designer, FDL_Designer）"
    },
    "version": {
      "type": "string",
      "pattern": "^\\d+\\.\\d+\\.\\d+$",
      "description": "事件 schema 版本（語義化版本號）"
    },
    "correlation_id": {
      "type": "string",
      "format": "uuid",
      "description": "關聯 ID，用於追蹤相關事件"
    },
    "seq": {
      "type": "integer",
      "minimum": 0,
      "description": "事件序列號，用於排序和去重"
    },
    "metadata": {
      "type": "object",
      "description": "額外的元數據"
    }
  }
}
```

2.2 核心事件類型定義

2.2.1 TagValueChanged 事件

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://idtf.org/schemas/events/tag-value-changed.json",
  "title": "Tag Value Changed Event",
  "description": "IOT Tag 數值變更事件",
  "allOf": [
    { "$ref": "base-event.json" }
  ],
  "type": "object",
  "required": [
    "asset_id",
    "tag_id",
    "value",
    "quality"
  ],
  "properties": {
    "event_type": {
      "const": "TagValueChanged"
    },
    "asset_id": {
      "type": "string",
      "description": "資產實例 ID"
    },
    "tag_id": {
      "type": "string",
      "description": "Tag ID"
    },
    "tag_name": {
      "type": "string",
      "description": "Tag 名稱"
    },
    "value": {
      "oneOf": [
        { "type": "number" },
        { "type": "string" },
        { "type": "boolean" }
      ],
      "description": "Tag 數值"
    },
    "previous_value": {
      "oneOf": [
        { "type": "number" },
        { "type": "string" },
        { "type": "boolean" },
        { "type": "null" }
      ],
      "description": "前一個數值"
    },
    "quality": {
      "type": "string",
      "enum": ["Good", "Bad", "Uncertain"],
      "description": "數據品質"
    },
    "unit": {
      "type": "string",
      "description": "單位（例如：°C, bar, RPM）"
    }
  }
}
```

範例：

```json
{
  "event_id": "550e8400-e29b-41d4-a716-446655440000",
  "event_type": "TagValueChanged",
  "timestamp": "2025-10-22T10:30:45.123Z",
  "source": "NDH",
  "version": "1.0.0",
  "seq": 12345,
  "asset_id": "pump_001",
  "tag_id": "tag_flow_001",
  "tag_name": "Flow",
  "value": 125.5,
  "previous_value": 120.3,
  "quality": "Good",
  "unit": "m³/h"
}
```

2.2.2 InstanceCreated 事件

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://idtf.org/schemas/events/instance-created.json",
  "title": "Instance Created Event",
  "description": "資產實例創建事件",
  "allOf": [
    { "$ref": "base-event.json" }
  ],
  "type": "object",
  "required": [
    "instance_id",
    "asset_id",
    "instance_type"
  ],
  "properties": {
    "event_type": {
      "const": "InstanceCreated"
    },
    "instance_id": {
      "type": "string",
      "description": "實例 ID"
    },
    "asset_id": {
      "type": "string",
      "description": "資產類型 ID"
    },
    "instance_type": {
      "type": "string",
      "enum": ["AssetInstance", "TagInstance"],
      "description": "實例類型"
    },
    "name": {
      "type": "string",
      "description": "實例名稱"
    },
    "position": {
      "type": "object",
      "properties": {
        "x": { "type": "number" },
        "y": { "type": "number" },
        "z": { "type": "number" }
      },
      "description": "3D 空間位置"
    },
    "properties": {
      "type": "object",
      "description": "實例屬性"
    }
  }
}
```

範例:

```json
{
  "event_id": "660e8400-e29b-41d4-a716-446655440001",
  "event_type": "InstanceCreated",
  "timestamp": "2025-10-22T10:25:30.456Z",
  "source": "FDL_Designer",
  "version": "1.0.0",
  "seq": 12340,
  "instance_id": "inst_pump_001",
  "asset_id": "asset_centrifugal_pump",
  "instance_type": "AssetInstance",
  "name": "Pump_1",
  "position": {
    "x": 10.5,
    "y": 0.0,
    "z": 5.2
  },
  "properties": {
    "manufacturer": "Grundfos",
    "model": "CR 64-2",
    "rated_power": 7.5
  }
}
```

2.2.3 AlarmRaised 事件

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://idtf.org/schemas/events/alarm-raised.json",
  "title": "Alarm Raised Event",
  "description": "警報觸發事件",
  "allOf": [
    { "$ref": "base-event.json" }
  ],
  "type": "object",
  "required": [
    "alarm_id",
    "asset_id",
    "tag_id",
    "severity",
    "message"
  ],
  "properties": {
    "event_type": {
      "const": "AlarmRaised"
    },
    "alarm_id": {
      "type": "string",
      "description": "警報 ID"
    },
    "asset_id": {
      "type": "string",
      "description": "資產實例 ID"
    },
    "tag_id": {
      "type": "string",
      "description": "Tag ID"
    },
    "severity": {
      "type": "string",
      "enum": ["Critical", "High", "Medium", "Low", "Info"],
      "description": "警報嚴重程度"
    },
    "message": {
      "type": "string",
      "description": "警報訊息"
    },
    "threshold": {
      "type": "number",
      "description": "觸發閾值"
    },
    "actual_value": {
      "type": "number",
      "description": "實際數值"
    }
  }
}
```

範例:

```json
{
  "event_id": "770e8400-e29b-41d4-a716-446655440002",
  "event_type": "AlarmRaised",
  "timestamp": "2025-10-22T10:31:15.789Z",
  "source": "NDH",
  "version": "1.0.0",
  "seq": 12346,
  "alarm_id": "alarm_temp_high_001",
  "asset_id": "pump_001",
  "tag_id": "tag_temp_001",
  "severity": "High",
  "message": "Temperature exceeded threshold",
  "threshold": 80.0,
  "actual_value": 85.3
}
```

2.2.4 其他事件類型

以下是其他常見事件類型的簡要定義：

事件類型	描述	主要欄位
InstanceDeleted	實例刪除	instance_id, asset_id
InstanceUpdated	實例更新	instance_id, changed_properties
ConnectionCreated	連接創建	connection_id, from_instance, to_instance
ConnectionDeleted	連接刪除	connection_id
CollisionDetected	碰撞檢測	instance_id_1, instance_id_2, overlap_volume
FDLLoaded	FDL 載入	fdl_file, instance_count, tag_count
TSDBConnected	TSDB 連接	tsdb_type, connection_string
TSDBDisconnected	TSDB 斷線	tsdb_type, reason

2.3 事件版本管理

事件 schema 使用語義化版本號（Semantic Versioning）：

- **MAJOR**: 不兼容的 schema 變更
- **MINOR**: 向後兼容的新增欄位
- **PATCH**: 向後兼容的 Bug 修復

範例： - `1.0.0` - 初始版本 - `1.1.0` - 新增 `correlation_id` 欄位（向後兼容） - `2.0.0` - 將 `timestamp` 從 Unix timestamp 改為 ISO 8601（不兼容）

---

3. 抽象 Event Bus 介面設計

3.1 IEventBus 介面

```python
# ndh/infrastructure/event_bus/i_event_bus.py

from abc import ABC, abstractmethod
from typing import Callable, Dict, Any, Optional, List
from enum import Enum

class DeliveryGuarantee(Enum):
    """事件傳遞保證級別"""
    AT_MOST_ONCE = "at_most_once"      # 最多一次（可能丟失）
    AT_LEAST_ONCE = "at_least_once"    # 至少一次（可能重複）
    EXACTLY_ONCE = "exactly_once"      # 恰好一次（最強保證）

class IEventBus(ABC):
    """Event Bus 抽象介面"""

    @abstractmethod
    def publish(
        self,
        event_type: str,
        event_data: Dict[str, Any],
        guarantee: DeliveryGuarantee = DeliveryGuarantee.AT_LEAST_ONCE
    ) -> bool:
        """
        發布事件

        Args:
            event_type: 事件類型
            event_data: 事件數據（必須符合對應的 JSON Schema）
            guarantee: 傳遞保證級別

        Returns:
            bool: 是否成功發布
        """
        pass

    @abstractmethod
    def subscribe(
        self,
        event_type: str,
        handler: Callable[[Dict[str, Any]], None],
        filter_expr: Optional[str] = None
    ) -> str:
        """
        訂閱事件

        Args:
            event_type: 事件類型（支援萬用字元，例如 "Tag*"）
            handler: 事件處理函式
            filter_expr: 過濾表達式（例如 "severity == 'High'"）

        Returns:
            str: 訂閱 ID
        """
        pass

    @abstractmethod
    def unsubscribe(self, subscription_id: str) -> bool:
        """
        取消訂閱

        Args:
            subscription_id: 訂閱 ID

        Returns:
            bool: 是否成功取消
        """
        pass

    @abstractmethod
    def start(self) -> bool:
        """啟動 Event Bus"""
        pass

    @abstractmethod
    def stop(self) -> bool:
        """停止 Event Bus"""
        pass

    @abstractmethod
    def get_event_history(
        self,
        event_type: Optional[str] = None,
        start_time: Optional[str] = None,
        end_time: Optional[str] = None,
        limit: int = 100
    ) -> List[Dict[str, Any]]:
        """
        獲取事件歷史（用於重放）

        Args:
            event_type: 事件類型過濾
            start_time: 開始時間（ISO 8601）
            end_time: 結束時間（ISO 8601）
            limit: 最大返回數量

        Returns:
            List[Dict]: 事件列表
        """
        pass

    @abstractmethod
    def replay_events(
        self,
        event_ids: List[str],
        target_handler: Optional[Callable] = None
    ) -> int:
        """
        重放事件

        Args:
            event_ids: 要重放的事件 ID 列表
            target_handler: 目標處理函式（如果為 None，則發送給所有訂閱者）

        Returns:
            int: 成功重放的事件數量
        """
        pass
```

3.2 In-Memory Event Bus 實作（MVP）

```python
# ndh/infrastructure/event_bus/in_memory_event_bus.py

import uuid
import threading
from datetime import datetime
from typing import Callable, Dict, Any, Optional, List
from collections import defaultdict
import fnmatch
import json

from .i_event_bus import IEventBus, DeliveryGuarantee

class InMemoryEventBus(IEventBus):
    """In-Memory Event Bus 實作（用於 MVP 和測試）"""

    def __init__(self, enable_persistence: bool = False):
        self._subscribers: Dict[str, List[Dict]] = defaultdict(list)
        self._event_history: List[Dict[str, Any]] = []
        self._lock = threading.RLock()
        self._running = False
        self._enable_persistence = enable_persistence
        self._seq_counter = 0

    def publish(
        self,
        event_type: str,
        event_data: Dict[str, Any],
        guarantee: DeliveryGuarantee = DeliveryGuarantee.AT_LEAST_ONCE
    ) -> bool:
        """發布事件"""
        if not self._running:
            return False

        with self._lock:
            # 添加基礎欄位
            if 'event_id' not in event_data:
                event_data['event_id'] = str(uuid.uuid4())
            if 'timestamp' not in event_data:
                event_data['timestamp'] = datetime.utcnow().isoformat() + 'Z'
            if 'seq' not in event_data:
                event_data['seq'] = self._seq_counter
                self._seq_counter += 1

            event_data['event_type'] = event_type

            # 持久化事件（如果啟用）
            if self._enable_persistence:
                self._event_history.append(event_data.copy())

            # 發送給訂閱者
            delivered_count = 0
            for pattern, subscribers in self._subscribers.items():
                if fnmatch.fnmatch(event_type, pattern):
                    for sub in subscribers:
                        try:
                            # 應用過濾器
                            if sub['filter_expr'] and not
self._eval_filter(event_data, sub['filter_expr']):
                                continue

                            sub['handler'](event_data.copy())
                            delivered_count += 1
                        except Exception as e:
                            print(f"Error delivering event to subscriber: {e}")

            return delivered_count > 0

    def subscribe(
        self,
        event_type: str,
        handler: Callable[[Dict[str, Any]], None],
        filter_expr: Optional[str] = None
    ) -> str:
        """訂閱事件"""
        subscription_id = str(uuid.uuid4())

        with self._lock:
            self._subscribers[event_type].append({
                'id': subscription_id,
                'handler': handler,
                'filter_expr': filter_expr
            })

        return subscription_id

    def unsubscribe(self, subscription_id: str) -> bool:
        """取消訂閱"""
        with self._lock:
            for event_type, subscribers in self._subscribers.items():
                self._subscribers[event_type] = [
                    sub for sub in subscribers if sub['id'] != subscription_id
                ]
            return True

    def start(self) -> bool:
        """啟動 Event Bus"""
        self._running = True
        return True

    def stop(self) -> bool:
        """停止 Event Bus"""
        self._running = False
        return True

    def get_event_history(
        self,
        event_type: Optional[str] = None,
        start_time: Optional[str] = None,
        end_time: Optional[str] = None,
        limit: int = 100
    ) -> List[Dict[str, Any]]:
        """獲取事件歷史"""
        with self._lock:
            filtered_events = self._event_history

            # 過濾事件類型
            if event_type:
                filtered_events = [
                    e for e in filtered_events
                    if fnmatch.fnmatch(e.get('event_type', ''), event_type)
                ]

            # 過濾時間範圍
            if start_time:
                filtered_events = [
                    e for e in filtered_events
                    if e.get('timestamp', '') >= start_time
                ]
            if end_time:
                filtered_events = [
                    e for e in filtered_events
                    if e.get('timestamp', '') <= end_time
                ]

            # 限制數量
            return filtered_events[-limit:]

    def replay_events(
        self,
        event_ids: List[str],
        target_handler: Optional[Callable] = None
    ) -> int:
        """重放事件"""
        with self._lock:
            replayed_count = 0

            for event in self._event_history:
                if event.get('event_id') in event_ids:
                    if target_handler:
                        target_handler(event.copy())
                    else:
                        self.publish(event['event_type'], event.copy())
                    replayed_count += 1

            return replayed_count

    def _eval_filter(self, event_data: Dict[str, Any], filter_expr: str) ->
bool:
        """評估過濾表達式（簡化版本）"""
        try:
            # 簡單的表達式評估（生產環境應使用更安全的方式）
            return eval(filter_expr, {"__builtins__": {}}, event_data)
        except:
            return True
```

3.3 ZMQ Event Bus 實作（擴充）

```python
# ndh/infrastructure/event_bus/zmq_event_bus.py

import zmq
import json
import threading
from typing import Callable, Dict, Any, Optional, List

from .i_event_bus import IEventBus, DeliveryGuarantee

class ZmqEventBus(IEventBus):
    """ZMQ Event Bus 實作（用於分散式部署）"""

    def __init__(self, pub_address: str = "tcp://*:5555", sub_address: str =
"tcp://localhost:5555"):
        self._pub_address = pub_address
        self._sub_address = sub_address
        self._context = zmq.Context()
        self._pub_socket = None
        self._sub_socket = None
        self._subscribers = {}
        self._running = False
        self._sub_thread = None

    def publish(
        self,
        event_type: str,
        event_data: Dict[str, Any],
        guarantee: DeliveryGuarantee = DeliveryGuarantee.AT_LEAST_ONCE
    ) -> bool:
        """發布事件"""
        if not self._running or not self._pub_socket:
            return False

        try:
            message = {
                'event_type': event_type,
                'data': event_data
            }

            # ZMQ 使用 topic-based pub/sub
            self._pub_socket.send_multipart([
                event_type.encode('utf-8'),
                json.dumps(message).encode('utf-8')
            ])

            return True
        except Exception as e:
            print(f"Error publishing event: {e}")
            return False

    def subscribe(
        self,
        event_type: str,
        handler: Callable[[Dict[str, Any]], None],
        filter_expr: Optional[str] = None
    ) -> str:
        """訂閱事件"""
        subscription_id = str(uuid.uuid4())

        self._subscribers[subscription_id] = {
            'event_type': event_type,
            'handler': handler,
            'filter_expr': filter_expr
        }

        # 訂閱 ZMQ topic
        if self._sub_socket:
            self._sub_socket.setsockopt_string(zmq.SUBSCRIBE, event_type)

        return subscription_id

    def unsubscribe(self, subscription_id: str) -> bool:
        """取消訂閱"""
        if subscription_id in self._subscribers:
            event_type = self._subscribers[subscription_id]['event_type']

            # 取消訂閱 ZMQ topic
            if self._sub_socket:
                self._sub_socket.setsockopt_string(zmq.UNSUBSCRIBE, event_type)

            del self._subscribers[subscription_id]
            return True

        return False

    def start(self) -> bool:
        """啟動 Event Bus"""
        try:
            # 創建 Publisher socket
            self._pub_socket = self._context.socket(zmq.PUB)
            self._pub_socket.bind(self._pub_address)

            # 創建 Subscriber socket
            self._sub_socket = self._context.socket(zmq.SUB)
            self._sub_socket.connect(self._sub_address)

            self._running = True

            # 啟動訂閱執行緒
            self._sub_thread = threading.Thread(target=self._subscription_loop,
daemon=True)
            self._sub_thread.start()

            return True
        except Exception as e:
            print(f"Error starting ZMQ Event Bus: {e}")
            return False

    def stop(self) -> bool:
        """停止 Event Bus"""
        self._running = False

        if self._sub_thread:
            self._sub_thread.join(timeout=1.0)

        if self._pub_socket:
            self._pub_socket.close()

        if self._sub_socket:
            self._sub_socket.close()

        return True

    def _subscription_loop(self):
        """訂閱循環（在獨立執行緒中運行）"""
        while self._running:
            try:
                # 接收訊息（非阻塞）
                if self._sub_socket.poll(timeout=100):
                    topic, message_bytes = self._sub_socket.recv_multipart()
                    message = json.loads(message_bytes.decode('utf-8'))

                    event_type = message['event_type']
                    event_data = message['data']

                    # 分發給訂閱者
                    for sub_id, sub in self._subscribers.items():
                        if fnmatch.fnmatch(event_type, sub['event_type']):
                            try:
                                sub['handler'](event_data)
                            except Exception as e:
                                print(f"Error in event handler: {e}")

            except Exception as e:
                print(f"Error in subscription loop: {e}")

    def get_event_history(self, *args, **kwargs) -> List[Dict[str, Any]]:
        """ZMQ 不支援事件歷史（需要額外的持久化層）"""
        return []

    def replay_events(self, *args, **kwargs) -> int:
        """ZMQ 不支援事件重放（需要額外的持久化層）"""
        return 0
```

3.4 MQTT Event Bus 實作（擴充）

```
# ndh/infrastructure/event_bus/mqtt_event_bus.py

import paho.mqtt.client as mqtt
import json
from typing import Callable, Dict, Any, Optional, List

from .i_event_bus import IEventBus, DeliveryGuarantee

class MqttEventBus(IEventBus):
    """MQTT Event Bus 實作（用於 IoT 場景）"""

    def __init__(self, broker_address: str = "localhost", port: int = 1883):
        self._broker_address = broker_address
        self._port = port
        self._client = mqtt.Client()
        self._subscribers = {}
        self._running = False

        # 設定 MQTT 回調
        self._client.on_connect = self._on_connect
        self._client.on_message = self._on_message

    def publish(
        self,
        event_type: str,
        event_data: Dict[str, Any],
        guarantee: DeliveryGuarantee = DeliveryGuarantee.AT_LEAST_ONCE
    ) -> bool:
        """發布事件"""
        if not self._running:
            return False

        try:
            topic = f"idtf/events/{event_type}"
            payload = json.dumps(event_data)

            # 根據保證級別設定 QoS
            qos = {
                DeliveryGuarantee.AT_MOST_ONCE: 0,
                DeliveryGuarantee.AT_LEAST_ONCE: 1,
                DeliveryGuarantee.EXACTLY_ONCE: 2
            }.get(guarantee, 1)

            result = self._client.publish(topic, payload, qos=qos)
            return result.rc == mqtt.MQTT_ERR_SUCCESS

        except Exception as e:
            print(f"Error publishing event: {e}")
            return False

    def subscribe(
        self,
        event_type: str,
        handler: Callable[[Dict[str, Any]], None],
        filter_expr: Optional[str] = None
    ) -> str:
        """訂閱事件"""
        subscription_id = str(uuid.uuid4())

        self._subscribers[subscription_id] = {
            'event_type': event_type,
            'handler': handler,
            'filter_expr': filter_expr
        }

        # 訂閱 MQTT topic
        topic = f"idtf/events/{event_type}"
        self._client.subscribe(topic, qos=1)

        return subscription_id

    def start(self) -> bool:
        """啟動 Event Bus"""
        try:
            self._client.connect(self._broker_address, self._port,
keepalive=60)
            self._client.loop_start()
            self._running = True
            return True
        except Exception as e:
            print(f"Error starting MQTT Event Bus: {e}")
            return False

    def stop(self) -> bool:
        """停止 Event Bus"""
        self._running = False
        self._client.loop_stop()
        self._client.disconnect()
        return True

    def _on_connect(self, client, userdata, flags, rc):
        """MQTT 連接回調"""
        if rc == 0:
            print("Connected to MQTT broker")
        else:
            print(f"Failed to connect to MQTT broker: {rc}")

    def _on_message(self, client, userdata, msg):
        """MQTT 訊息回調"""
        try:
            event_data = json.loads(msg.payload.decode('utf-8'))
            event_type = msg.topic.split('/')[-1]

            # 分發給訂閱者
            for sub_id, sub in self._subscribers.items():
                if fnmatch.fnmatch(event_type, sub['event_type']):
                    try:
                        sub['handler'](event_data)
                    except Exception as e:
                        print(f"Error in event handler: {e}")

        except Exception as e:
            print(f"Error processing MQTT message: {e}")

    def get_event_history(self, *args, **kwargs) -> List[Dict[str, Any]]:
        """MQTT 不支援事件歷史（需要額外的持久化層）"""
        return []

    def replay_events(self, *args, **kwargs) -> int:
        """MQTT 不支援事件重放（需要額外的持久化層）"""
        return 0
```

4. TSDB 抽象與演進路徑

4.1 ITSDBService 介面

```python
# ndh/infrastructure/tsdb/i_tsdb_service.py

from abc import ABC, abstractmethod
from typing import List, Dict, Any, Optional
from datetime import datetime

class ITSDBService(ABC):
    """時序數據庫抽象介面"""

    @abstractmethod
    def connect(self, connection_string: str, **kwargs) -> bool:
        """
        連接到 TSDB

        Args:
            connection_string: 連接字串
            **kwargs: 額外參數

        Returns:
            bool: 是否成功連接
        """
        pass

    @abstractmethod
    def disconnect(self) -> bool:
        """斷開連接"""
        pass

    @abstractmethod
    def write_single(
        self,
        tag_id: str,
        timestamp: datetime,
        value: Any,
        quality: str = "Good",
        metadata: Optional[Dict[str, Any]] = None
    ) -> bool:
        """
        寫入單個數據點

        Args:
            tag_id: Tag ID
            timestamp: 時間戳
            value: 數值
            quality: 數據品質
            metadata: 額外元數據

        Returns:
            bool: 是否成功寫入
        """
        pass

    @abstractmethod
    def write_batch(
        self,
        data_points: List[Dict[str, Any]]
    ) -> int:
        """
        批量寫入數據點

        Args:
            data_points: 數據點列表，每個數據點包含 tag_id, timestamp, value,
quality

        Returns:
            int: 成功寫入的數據點數量
        """
        pass

    @abstractmethod
    def read_latest(
        self,
        tag_ids: List[str]
    ) -> Dict[str, Dict[str, Any]]:
        """
        讀取最新數據

        Args:
            tag_ids: Tag ID 列表

        Returns:
            Dict: {tag_id: {timestamp, value, quality}}
        """
        pass

    @abstractmethod
    def read_range(
        self,
        tag_ids: List[str],
        start_time: datetime,
        end_time: datetime,
        limit: Optional[int] = None
    ) -> Dict[str, List[Dict[str, Any]]]:
        """
        讀取時間範圍內的數據

        Args:
            tag_ids: Tag ID 列表
            start_time: 開始時間
            end_time: 結束時間
            limit: 最大返回數量

        Returns:
            Dict: {tag_id: [{timestamp, value, quality}, ...]}
        """
        pass

    @abstractmethod
    def aggregate(
        self,
        tag_id: str,
        start_time: datetime,
        end_time: datetime,
        aggregation: str,  # "avg", "min", "max", "sum", "count"
        interval: Optional[str] = None  # "1m", "5m", "1h", "1d"
    ) -> List[Dict[str, Any]]:
        """
        聚合查詢

        Args:
            tag_id: Tag ID
            start_time: 開始時間
            end_time: 結束時間
            aggregation: 聚合函式
            interval: 聚合間隔

        Returns:
            List: [{timestamp, value}, ...]
        """
        pass

    @abstractmethod
    def get_statistics(self) -> Dict[str, Any]:
        """
        獲取 TSDB 統計資訊

        Returns:
            Dict: {tag_count, data_point_count, storage_size, ...}
        """
        pass
```

4.2 SQLite TSDB 實作（MVP）

```python
# ndh/infrastructure/tsdb/sqlite_tsdb_service.py

import sqlite3
from datetime import datetime
from typing import List, Dict, Any, Optional
import json

from .i_tsdb_service import ITSDBService

class SqliteTSDBService(ITSDBService):
    """SQLite TSDB 實作（用於 MVP 和本地開發）"""

    def __init__(self):
        self._conn: Optional[sqlite3.Connection] = None
        self._cursor: Optional[sqlite3.Cursor] = None

    def connect(self, connection_string: str, **kwargs) -> bool:
        """連接到 SQLite 數據庫"""
        try:
            self._conn = sqlite3.connect(connection_string,
check_same_thread=False)
            self._cursor = self._conn.cursor()

            # 創建表
            self._cursor.execute('''
                CREATE TABLE IF NOT EXISTS time_series_data (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    tag_id TEXT NOT NULL,
                    timestamp TEXT NOT NULL,
                    value REAL,
                    value_text TEXT,
                    quality TEXT DEFAULT 'Good',
                    metadata TEXT,
                    created_at TEXT DEFAULT CURRENT_TIMESTAMP
                )
            ''')

            # 創建索引
            self._cursor.execute('''
                CREATE INDEX IF NOT EXISTS idx_tag_timestamp
                ON time_series_data(tag_id, timestamp)
            ''')

            self._conn.commit()
            return True

        except Exception as e:
            print(f"Error connecting to SQLite: {e}")
            return False

    def disconnect(self) -> bool:
        """斷開連接"""
        if self._conn:
            self._conn.close()
            return True
        return False

    def write_single(
        self,
        tag_id: str,
        timestamp: datetime,
        value: Any,
        quality: str = "Good",
        metadata: Optional[Dict[str, Any]] = None
    ) -> bool:
        """寫入單個數據點"""
        try:
            timestamp_str = timestamp.isoformat()

            # 根據值類型選擇欄位
            if isinstance(value, (int, float)):
                value_num = float(value)
                value_text = None
            else:
                value_num = None
                value_text = str(value)

            metadata_json = json.dumps(metadata) if metadata else None

            self._cursor.execute('''
                INSERT INTO time_series_data (tag_id, timestamp, value,
value_text, quality, metadata)
                VALUES (?, ?, ?, ?, ?, ?)
            ''', (tag_id, timestamp_str, value_num, value_text, quality,
metadata_json))

            self._conn.commit()
            return True

        except Exception as e:
            print(f"Error writing data point: {e}")
            return False

    def write_batch(self, data_points: List[Dict[str, Any]]) -> int:
        """批量寫入數據點"""
        success_count = 0

        try:
            for dp in data_points:
                timestamp_str = dp['timestamp'].isoformat() if
isinstance(dp['timestamp'], datetime) else dp['timestamp']

                if isinstance(dp['value'], (int, float)):
                    value_num = float(dp['value'])
                    value_text = None
                else:
                    value_num = None
                    value_text = str(dp['value'])

                metadata_json = json.dumps(dp.get('metadata')) if
dp.get('metadata') else None

                self._cursor.execute('''
                    INSERT INTO time_series_data (tag_id, timestamp, value,
value_text, quality, metadata)
                    VALUES (?, ?, ?, ?, ?, ?)
                ''', (
                    dp['tag_id'],
                    timestamp_str,
                    value_num,
                    value_text,
                    dp.get('quality', 'Good'),
                    metadata_json
                ))

                success_count += 1

            self._conn.commit()
        except Exception as e:
            print(f"Error in batch write: {e}")
            self._conn.rollback()

        return success_count

    def read_latest(self, tag_ids: List[str]) -> Dict[str, Dict[str, Any]]:
        """讀取最新數據"""
        result = {}

        for tag_id in tag_ids:
            self._cursor.execute('''
                SELECT timestamp, value, value_text, quality, metadata
                FROM time_series_data
                WHERE tag_id = ?
                ORDER BY timestamp DESC
                LIMIT 1
            ''', (tag_id,))

            row = self._cursor.fetchone()
            if row:
                result[tag_id] = {
                    'timestamp': row[0],
                    'value': row[1] if row[1] is not None else row[2],
                    'quality': row[3],
                    'metadata': json.loads(row[4]) if row[4] else None
                }

        return result

    def read_range(
        self,
        tag_ids: List[str],
        start_time: datetime,
        end_time: datetime,
        limit: Optional[int] = None
    ) -> Dict[str, List[Dict[str, Any]]]:
        """讀取時間範圍內的數據"""
        result = {tag_id: [] for tag_id in tag_ids}

        start_str = start_time.isoformat()
        end_str = end_time.isoformat()

        for tag_id in tag_ids:
            query = '''
                SELECT timestamp, value, value_text, quality
                FROM time_series_data
                WHERE tag_id = ? AND timestamp >= ? AND timestamp <= ?
                ORDER BY timestamp ASC
            '''

            if limit:
                query += f' LIMIT {limit}'

            self._cursor.execute(query, (tag_id, start_str, end_str))

            for row in self._cursor.fetchall():
                result[tag_id].append({
                    'timestamp': row[0],
                    'value': row[1] if row[1] is not None else row[2],
                    'quality': row[3]
                })

        return result
    def aggregate(
        self,
        tag_id: str,
        start_time: datetime,
        end_time: datetime,
        aggregation: str,
        interval: Optional[str] = None
    ) -> List[Dict[str, Any]]:
        """聚合查詢（簡化版本）"""
        start_str = start_time.isoformat()
        end_str = end_time.isoformat()

        # SQLite 的聚合函式
        agg_func = {
            'avg': 'AVG',
            'min': 'MIN',
            'max': 'MAX',
            'sum': 'SUM',
            'count': 'COUNT'
        }.get(aggregation.lower(), 'AVG')

        if interval:
            # 簡化版本：不支援時間間隔聚合
            # 生產環境應使用 DuckDB 或專門的 TSDB
            pass

        query = f'''
            SELECT {agg_func}(value) as agg_value
            FROM time_series_data
            WHERE tag_id = ? AND timestamp >= ? AND timestamp <= ?
        '''

        self._cursor.execute(query, (tag_id, start_str, end_str))
        row = self._cursor.fetchone()

        if row and row[0] is not None:
            return [{
                'timestamp': end_str,
                'value': row[0]
            }]

        return []

    def get_statistics(self) -> Dict[str, Any]:
        """獲取統計資訊"""
        stats = {}

        # Tag 數量
        self._cursor.execute('SELECT COUNT(DISTINCT tag_id) FROM 
time_series_data')
        stats['tag_count'] = self._cursor.fetchone()[0]

        # 數據點數量
        self._cursor.execute('SELECT COUNT(*) FROM time_series_data')
        stats['data_point_count'] = self._cursor.fetchone()[0]

        # 數據庫大小（頁數 * 頁大小）
        self._cursor.execute('PRAGMA page_count')
        page_count = self._cursor.fetchone()[0]
        self._cursor.execute('PRAGMA page_size')
        page_size = self._cursor.fetchone()[0]
        stats['storage_size_bytes'] = page_count * page_size

        return stats
```

4.3 DuckDB/Parquet TSDB 實作（P1）

```
# ndh/infrastructure/tsdb/duckdb_tsdb_service.py

import duckdb
from datetime import datetime
from typing import List, Dict, Any, Optional
import pandas as pd

from .i_tsdb_service import ITSDBService

class DuckDBTSDBService(ITSDBService):
    """DuckDB TSDB 實作（用於離線分析和大數據查詢）"""

    def __init__(self):
        self._conn: Optional[duckdb.DuckDBPyConnection] = None
        self._parquet_path: Optional[str] = None

    def connect(self, connection_string: str, **kwargs) -> bool:
        """連接到 DuckDB"""
        try:
            self._conn = duckdb.connect(connection_string)
            self._parquet_path = kwargs.get('parquet_path',
'./data/tsdb.parquet')

            # 創建表（如果不存在）
            self._conn.execute('''
                CREATE TABLE IF NOT EXISTS time_series_data (
                    tag_id VARCHAR,
                    timestamp TIMESTAMP,
                    value DOUBLE,
                    value_text VARCHAR,
                    quality VARCHAR,
                    metadata JSON
                )
            ''')

            # 如果 Parquet 檔案存在，載入數據
            try:
                self._conn.execute(f'''
                    INSERT INTO time_series_data
                    SELECT * FROM read_parquet('{self._parquet_path}')
                ''')
            except:
                pass  # Parquet 檔案不存在

            return True

        except Exception as e:
            print(f"Error connecting to DuckDB: {e}")
            return False

    def disconnect(self) -> bool:
        """斷開連接並儲存到 Parquet"""
        if self._conn:
            try:
                # 匯出到 Parquet
                self._conn.execute(f'''
                    COPY time_series_data TO '{self._parquet_path}' (FORMAT
PARQUET)
                ''')
            except Exception as e:
                print(f"Error exporting to Parquet: {e}")
                self._conn.close()
                return True
            return False

    def write_batch(self, data_points: List[Dict[str, Any]]) -> int:
        """批量寫入（DuckDB 優化）"""
        try:
            # 使用 Pandas DataFrame 進行批量插入
            df = pd.DataFrame(data_points)

            # 轉換時間戳
            if 'timestamp' in df.columns:
                df['timestamp'] = pd.to_datetime(df['timestamp'])

            # 插入到 DuckDB
            self._conn.execute('INSERT INTO time_series_data SELECT * FROM df')

            return len(data_points)

        except Exception as e:
            print(f"Error in batch write: {e}")
            return 0

    def aggregate(
        self,
        tag_id: str,
        start_time: datetime,
        end_time: datetime,
        aggregation: str,
        interval: Optional[str] = None
    ) -> List[Dict[str, Any]]:
        """聚合查詢（DuckDB 優化，支援時間間隔）"""
        agg_func = aggregation.upper()

        if interval:
            # 解析間隔（例如 "1m", "5m", "1h", "1d"）
            interval_map = {
                'm': 'MINUTE',
                'h': 'HOUR',
                'd': 'DAY'
            }

            interval_value = int(interval[:-1])
            interval_unit = interval_map.get(interval[-1], 'MINUTE')

            query = f'''
                SELECT
                    time_bucket(INTERVAL '{interval_value} {interval_unit}',
timestamp) as bucket,
                    {agg_func}(value) as agg_value
                FROM time_series_data
                WHERE tag_id = ? AND timestamp >= ? AND timestamp <= ?
                GROUP BY bucket
                ORDER BY bucket
            '''
        else:
            query = f'''
                SELECT
                    ? as timestamp,
                    {agg_func}(value) as agg_value
                FROM time_series_data
                WHERE tag_id = ? AND timestamp >= ? AND timestamp <= ?
            '''

        result = self._conn.execute(query, [tag_id, start_time,
end_time]).fetchall()
```

```
        return [
            {'timestamp': row[0], 'value': row[1]}
            for row in result if row[1] is not None
        ]

    # 其他方法與 SQLite 實作類似...
```

4.4 TSDB 演進路徑

階段	TSDB 方案	適用場景	優勢	限制
MVP	**SQLite**	本地開發、小規模測試	• 零配置 • 跨平台 • 易於調試	• 寫入效能有限 • 不支援分散式 • 時序查詢不優化
P1	**DuckDB + Parquet**	離線分析、大數據查詢	• 優秀的分析效能 • 支援 Parquet 格式 • 豐富的 SQL 功能	• 主要用於離線分析 • 實時寫入效能一般
P2	**TDEngine**	生產環境、大規模部署	• 專為時序數據設計 • 高效能寫入和查詢 • 支援分散式	• 需要額外部署 • 學習曲線
P2	**InfluxDB**	雲端部署、IoT 場景	• 成熟的 TSDB • 豐富的生態系統 • 支援 Flux 查詢語言	• 企業版收費 • 資源消耗較高
P2	**AVEVA PI Adapter**	企業整合、工業場景	• 與 AVEVA PI 無縫整合 • 工業標準 • 強大的歷史數據管理	• 商業軟體 • 需要 PI Server

4.5 TSDB 切換策略

由於使用了 `ITSDBService` 抽象介面，切換 TSDB 實作非常簡單：

```python
# ndh/core/ndh_core.py

from ndh.infrastructure.tsdb.i_tsdb_service import ITSDBService
from ndh.infrastructure.tsdb.sqlite_tsdb_service import SqliteTSDBService
from ndh.infrastructure.tsdb.duckdb_tsdb_service import DuckDBTSDBService
# from ndh.infrastructure.tsdb.tdengine_tsdb_service import TDEngineTSDBService

class NDHCore:
    def __init__(self, config: Dict[str, Any]):
        # 根據配置選擇 TSDB 實作
        tsdb_type = config.get('tsdb_type', 'sqlite')

        if tsdb_type == 'sqlite':
            self.tsdb_service: ITSDBService = SqliteTSDBService()
        elif tsdb_type == 'duckdb':
            self.tsdb_service: ITSDBService = DuckDBTSDBService()
        elif tsdb_type == 'tdengine':
            self.tsdb_service: ITSDBService = TDEngineTSDBService()
        else:
            raise ValueError(f"Unsupported TSDB type: {tsdb_type}")

        # 連接到 TSDB
        connection_string = config.get('tsdb_connection_string',
'./data/tsdb.db')
        self.tsdb_service.connect(connection_string)
```

配置檔案範例：

```yaml
# config/ndh_config.yaml

# MVP 配置
tsdb_type: sqlite
tsdb_connection_string: ./data/tsdb.db

# P1 配置（切換到 DuckDB）
# tsdb_type: duckdb
# tsdb_connection_string: ./data/tsdb.duckdb
# parquet_path: ./data/tsdb.parquet

# P2 配置（切換到 TDEngine）
# tsdb_type: tdengine
# tsdb_connection_string:
host=localhost;port=6030;user=root;password=taosdata;database=idtf
```

---

5. 事件持久化與重放

5.1 事件日誌設計

為了支援事件重放和審計，我們需要一個事件日誌（Event Log）：

```python
# ndh/infrastructure/event_bus/event_logger.py

import sqlite3
import json
from datetime import datetime
from typing import Dict, Any, List, Optional

class EventLogger:
    """事件日誌（用於持久化和重放）"""

    def __init__(self, db_path: str = "./data/event_log.db"):
        self._conn = sqlite3.connect(db_path, check_same_thread=False)
        self._cursor = self._conn.cursor()

        # 創建事件日誌表
        self._cursor.execute('''
            CREATE TABLE IF NOT EXISTS event_log (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                event_id TEXT UNIQUE NOT NULL,
                event_type TEXT NOT NULL,
                timestamp TEXT NOT NULL,
                source TEXT,
                seq INTEGER,
                payload TEXT NOT NULL,
                created_at TEXT DEFAULT CURRENT_TIMESTAMP
            )
        ''')

        # 創建索引
        self._cursor.execute('''
            CREATE INDEX IF NOT EXISTS idx_event_type_timestamp
            ON event_log(event_type, timestamp)
        ''')

        self._cursor.execute('''
            CREATE INDEX IF NOT EXISTS idx_seq
            ON event_log(seq)
        ''')

        self._conn.commit()

    def log_event(self, event_data: Dict[str, Any]) -> bool:
        """記錄事件"""
        try:
            self._cursor.execute('''
                INSERT OR REPLACE INTO event_log (event_id, event_type,
timestamp, source, seq, payload)
                VALUES (?, ?, ?, ?, ?, ?)
            ''', (
                event_data.get('event_id'),
                event_data.get('event_type'),
                event_data.get('timestamp'),
                event_data.get('source'),
                event_data.get('seq'),
                json.dumps(event_data)
            ))

            self._conn.commit()
            return True

        except Exception as e:
            print(f"Error logging event: {e}")
            return False

    def get_events(
        self,
        event_type: Optional[str] = None,
        start_time: Optional[str] = None,
        end_time: Optional[str] = None,
        limit: int = 100
    ) -> List[Dict[str, Any]]:
        """查詢事件"""
        query = 'SELECT payload FROM event_log WHERE 1=1'
        params = []

        if event_type:
            query += ' AND event_type = ?'
            params.append(event_type)

        if start_time:
            query += ' AND timestamp >= ?'
            params.append(start_time)

        if end_time:
            query += ' AND timestamp <= ?'
            params.append(end_time)

        query += ' ORDER BY seq ASC LIMIT ?'
        params.append(limit)

        self._cursor.execute(query, params)

        return [json.loads(row[0]) for row in self._cursor.fetchall()]

    def get_events_by_ids(self, event_ids: List[str]) -> List[Dict[str, Any]]:
        """根據 ID 查詢事件"""
        placeholders = ','.join('?' * len(event_ids))
        query = f'SELECT payload FROM event_log WHERE event_id IN
({placeholders}) ORDER BY seq ASC'

        self._cursor.execute(query, event_ids)

        return [json.loads(row[0]) for row in self._cursor.fetchall()]
```

5.2 整合事件日誌到 Event Bus

```
# ndh/infrastructure/event_bus/in_memory_event_bus.py (更新)

class InMemoryEventBus(IEventBus):
    def __init__(self, enable_persistence: bool = False, event_logger: 
Optional[EventLogger] = None):
        # ... 原有初始化 ...
        self._event_logger = event_logger

    def publish(self, event_type: str, event_data: Dict[str, Any], guarantee: 
DeliveryGuarantee = DeliveryGuarantee.AT_LEAST_ONCE) -> bool:
        # ... 原有邏輯 ...

        # 記錄事件到日誌
        if self._event_logger:
            self._event_logger.log_event(event_data)

        # ... 原有邏輯 ...

    def get_event_history(self, event_type: Optional[str] = None, start_time: 
Optional[str] = None, end_time: Optional[str] = None, limit: int = 100) -> 
List[Dict[str, Any]]:
        """從事件日誌獲取歷史"""
        if self._event_logger:
            return self._event_logger.get_events(event_type, start_time, 
end_time, limit)
        else:
            return self._event_history[-limit:]

    def replay_events(self, event_ids: List[str], target_handler: 
Optional[Callable] = None) -> int:
        """從事件日誌重放事件"""
        if self._event_logger:
            events = self._event_logger.get_events_by_ids(event_ids)
        else:
            events = [e for e in self._event_history if e.get('event_id') in 
event_ids]

        replayed_count = 0
        for event in events:
            if target_handler:
                target_handler(event.copy())
            else:
                self.publish(event['event_type'], event.copy())
            replayed_count += 1

        return replayed_count
```

6. 使用範例

6.1 發布和訂閱事件

```python
from ndh.infrastructure.event_bus.in_memory_event_bus import InMemoryEventBus
from ndh.infrastructure.event_bus.event_logger import EventLogger

# 創建 Event Bus（啟用持久化）
event_logger = EventLogger("./data/event_log.db")
event_bus = InMemoryEventBus(enable_persistence=True,
event_logger=event_logger)
event_bus.start()

# 訂閱 TagValueChanged 事件
def on_tag_value_changed(event_data):
    print(f"Tag {event_data['tag_name']} changed to {event_data['value']}")

subscription_id = event_bus.subscribe("TagValueChanged", on_tag_value_changed)

# 發布事件
event_bus.publish("TagValueChanged", {
    "asset_id": "pump_001",
    "tag_id": "tag_flow_001",
    "tag_name": "Flow",
    "value": 125.5,
    "previous_value": 120.3,
    "quality": "Good",
    "unit": "m³/h",
    "source": "NDH",
    "version": "1.0.0"
})

# 查詢事件歷史
history = event_bus.get_event_history(event_type="TagValueChanged", limit=10)
print(f"Found {len(history)} events")

# 重放事件
event_ids = [event['event_id'] for event in history[:5]]
replayed_count = event_bus.replay_events(event_ids)
print(f"Replayed {replayed_count} events")
```

6.2 使用 TSDB

```
from ndh.infrastructure.tsdb.sqlite_tsdb_service import SqliteTSDBService
from datetime import datetime

# 創建 TSDB Service
tsdb = SqliteTSDBService()
tsdb.connect("./data/tsdb.db")

# 寫入單個數據點
tsdb.write_single(
    tag_id="tag_flow_001",
    timestamp=datetime.utcnow(),
    value=125.5,
    quality="Good"
)

# 批量寫入
data_points = [
    {"tag_id": "tag_flow_001", "timestamp": datetime.utcnow(), "value": 126.0,
"quality": "Good"},
    {"tag_id": "tag_temp_001", "timestamp": datetime.utcnow(), "value": 75.3,
"quality": "Good"},
]
tsdb.write_batch(data_points)

# 讀取最新數據
latest = tsdb.read_latest(["tag_flow_001", "tag_temp_001"])
print(latest)

# 聚合查詢
from datetime import timedelta
end_time = datetime.utcnow()
start_time = end_time - timedelta(hours=1)

avg_flow = tsdb.aggregate(
    tag_id="tag_flow_001",
    start_time=start_time,
    end_time=end_time,
    aggregation="avg"
)
print(f"Average flow: {avg_flow}")
```

---

7. 總結

本設計文件提供了完整的 NDH 事件模型與 TSDB 選型方案：

7.1 核心改進

✅ **事件契約標準化**：使用 JSON Schema 定義所有事件類型
✅ **抽象 Event Bus 介面**：支援 In-Memory, ZMQ, MQTT 多種實作
✅ **TSDB 抽象與演進**：SQLite → DuckDB → TDEngine/InfluxDB/AVEVA PI

✅ **事件持久化與重放**：支援事件日誌和審計
✅ **事件重送與可靠性**：支援 at-least-once 和 exactly-once 語義

7.2 演進路徑

階段	Event Bus	TSDB	適用場景
MVP	In-Memory	SQLite	本地開發、概念驗證
P1	ZMQ / MQTT	DuckDB + Parquet	分散式測試、離線分析
P2	Kafka	TDEngine / InfluxDB / AVEVA PI	生產環境、大規模部署

7.3 切換成本

由於採用了抽象介面設計，切換 Event Bus 和 TSDB 實作的成本極低：

- **Event Bus 切換**：修改配置檔案，無需修改業務邏輯
- **TSDB 切換**：修改配置檔案，無需修改數據存取代碼
- **向後兼容**：事件 schema 使用語義化版本號，支援平滑升級

這種設計確保了系統的靈活性和可擴展性，為未來的演進提供了堅實的基礎。

文件版本: 1.1
最後更新: 2025-10-22
作者: Michael Lin 林志錚
組織: HTFA/Digital Twins Alliance