Performative transactions: worlding compositional ecosystems

Lukawski, A.

Chapter 6. Decentralised Creative Networks and Performative Transactions

Decentralised Creative Networks

This chapter introduces the design and implementation of Decentralised Creative Networks—blockchain-based ecosystems where human artists and AI agents can transparently build upon each other's interoperable contributions (first discussed in Łukawski, 2024, 36–37). By sharing composable processes encoded as smart contracts, they form an evolving catalogue of artistic logics that can be activated and re-contextualised through executions called Performative Transactions. Each transaction constitutes a referential gesture: a decision to inherit, transform, and extend prior contributions, and thus to participate in a cumulative network of artistic activity.

The term "Creative Networks" was first theorised by Rasa Smite to describe collaborative artistic practices based on the early Internet in the 1990s—projects that explored new forms of social communication, artistic self-expression, and experimental infrastructure through open servers and shared protocols (Smite, 2012; cited in Łukawski, 2024, 38–39). Decentralised Creative Networks echoe this spirit of collaborative infrastructure, but reorient it through the affordances of blockchain and decentralised technology—particularly their capacity to encode modularity, composability, traceability, and programmability at the level of operations.

As Future Art Ecosystems 3 notes, this kind of modular logic—one that supports "permissionless participation, composable functionality and interoperability" (Serpentine Arts Technologies, 2022, 63)—is foundational not only to the ideals of decentralised culture, but to the building of new institutional imaginaries: an approach that they describe as "a GitHub for the arts" (2022, 104). Within such a system, authorship is no longer a fixed identity but becomes a function within a network of contributions. The "modularisation of identity" (136) enables contributors, whether human or AI, to operate across projects and contexts without collapsing into singular attribution. Instead of treating provenance as an after-the-fact marker of ownership, the system integrates provenance as a core operational principle—a dynamic ledger of artistic logic and agency embedded in every compositional transaction.

On the technical side, the concept answers a persistent challenge in generative blockchain-based art: the absence of standards for cross-application composability.

While the original intention behind the invention of smart contracts is to encode modular, re-usable logic (Ethereum, 2023), most current generative art NFTs are designed to operate within closed application contexts. They mirror the architectures of centralised software by linking to standardised file formats (such as audio, MIDI, and image files) that fail to represent the structural relations preceding the creation of artistic structures, like how musical intervals and rhythms build more intricate chords, melodies, forms, movements, and textures, without the flattening of such information within the final signal that becomes a media file (Łukawski, 2024, 39–41).

The system described here encodes relations between artistic operations. It functions more like a compositional operating system, or in a spirit envisioned by Claudio Tessone (2024) for future blockchain applications: a Xanadu-like environment for executable provenance, "a system that intricately links artistic assets to their origins and maintains a record of their history and transactions" (93). Project Xanadu was a visionary digital publishing system proposed by Ted Nelson in the 1960s. Unlike the modern web, which uses one-way links and often loses track of changes or authorship, Xanadu was designed to connect documents in both directions, preserve the full history of edits, and show exactly where every piece of content came from. It aimed to support transparent collaboration, shared ownership, and long-term traceability of digital material. A key concept in Xanadu was *transclusion*—the ability to include parts of one document within another while retaining a live connection to the original, so that any reuse of content always shows its source and context. As further observed by Tessone, a blockchain-based system inspired by Ted Nelson's vision of transclusion and versioned interlinking would not only enhance attribution but propose new models for redistribution, licensing, and collaborative inheritance—offering "a unique solution yet to be fully realised in the digital art world" (Tessone, 2024, 93).

The primary operation of the Decentralised Creative Network, addressing the need for introducing a standardised format for referencing, inheriting, and executing compositional logic across contributors and applications, is operationalised here in the concept of Performative Transaction. Performative Transactions, as explained further in this chapter, are smart contracts that model the logic of recombination and execution itself. This allows for the composition of processes in a way that preserves traceability, modularity, and interpretive flexibility.

As Artemi-Maria Gioti (2021) has argued, the development of tools for creative and artistic practices must be understood as an inherently aesthetic process. The design of

such systems is more than a neutral technical act. It is a constitutively artistic one. "All design decisions are—and should be viewed as—inherently aesthetic", she writes, underscoring the need for closer collaboration between composers and developers when building systems that support open-ended artistic exploration (144). This is particularly crucial in domains involving artificial intelligence or algorithmic systems, where the default orientation of available tools may reflect assumptions alien to the needs of contemporary artistic practice. To realign technological development with the aesthetic and methodological concerns of composers, Gioti calls for the involvement of artists as co-designers, shaping both the functionality and conceptual framing of the tools they engage with.

Following this principle, the technical realisation of the concepts of the Decentralised Creative Network and its core mechanism—Performative Transactions—that I proposed during this doctoral trajectory, was developed through a sustained collaboration between myself and programmer Michał Skarzyński. From September 2024 to May 2025, we worked closely in a process involving iterative design, implementation, and testing to build the working infrastructure described in this chapter. While I was responsible for the original architectural vision, the conceptual framework, and the formal requirements for compositional functionality, Skarzyński fully implemented this framework in code—writing smart contracts, designing the server infrastructure, expanding multiple low-level technical details and ensuring that the system functioned according to the recursive, modular logic of the compositional model. Every instance of software mentioned in this chapter—the public API and the development server—was implemented by Michał Skarzyński based on a compositional framework I envisioned, and has since been continuously refined through our ongoing collaboration. As the next sections will show, this implementation transforms Performative Transactions from a conceptual proposal into a model for decentralised compositional practice—capable of encoding not only the generation of artistic outputs but the rules, processes, and relations that structure their emergence.

A Compositional System as Public API

Decentralised Creative Network (DCN) is a system implemented as a public, open-sourced server code which, once installed, exposes API endpoints allowing for

interaction with the underlying, blockchain-based smart contracts via command-line tools, web interfaces, or by using automated AI agents, to contribute, retrieve, execute, and recombine compositional processes. The code of the server implementation is open-source, available as a public GitHub repository and can be installed locally by anyone, enabling direct interaction with the system without relying on any third-party infrastructure.

An API, or Application Programming Interface, is a structured way for one program (or user) to interact with another system. APIs are common in everyday software: for example, when a weather app displays forecasts, it is likely using a weather API that provides temperature and condition data from an external source. When a website embeds a Spotify playlist, it does so through the Spotify API, which exposes musical content and playback functions in a standardised way. The API introduced here functions in a similar way. It is a set of compositional functions for interacting with the blockchain network. These functions allow anyone, including AI agents, to create and execute reusable compositional processes encoded on the blockchain as part of a growing shared library of interdependent compositional processes.

Developers can build their own applications on top of this API. Advanced users can build intricate decentralised artistic ecosystems by encoding their envisioned artistic logics with no further care about the technicalities of lower level blockchain programming logic, while artists without prior programming experience can re-use the logics built by others to compose more intricate ecosystems from re-usable components. Most importantly, designing the compositional system as a public API introduces the possibility of composing self-reconfiguring systems—in which contributors modify the very compositional environment they are working within.

Composing with Transformations, Features, and Conditions

DCN allows its users to compose artistic processes by assembling three types of reusable building blocks: transformations, features, and conditions. Let's discuss them one-by-one:

Transformations are programmable functions that take numerical inputs and return outputs. They can be thought of as atomic musical operations—for example, "transpose pitch by 2", "multiply duration by 3", or more advanced logic such as "ask

an AI model to perform an action and return a number". Each transformation is implemented as a small Solidity function on the Ethereum blockchain, but users do not need to handle blockchain logic themselves since this is automated for them in the DCN—only the core operation (e.g. return x + 1). Once published, a transformation becomes a reusable building block that others can incorporate into their own processes. Most users will rely on an expanding library of existing transformations, while advanced users may create entirely new ones when necessary.

Features are the primary building blocks used to compose artistic logic within the DCN. A feature defines how a sequence of operations unfolds by combining existing transformations into an ordered structure. When executed, a feature applies its internal operations in a cyclic manner, looping back to the beginning once it reaches the end. For example, a feature encoding the steps of a pitch scale could continue indefinitely into higher registers. While each feature is a self-contained sequence of transformations, features can be combined into larger structures through Performative Transactions that link them together. Once published, they become reusable, shareable, and composable units that others can build upon when designing their own artistic processes.

Conditions define whether a feature can be executed. Some conditions are simple (e.g., "always true"), while others can be more complex—such as artistic logic (e.g., "execute only if the colour of this object is yellow"), time-based logic (e.g., "execute only between 12 am and 12 pm"), counters (e.g., "execute only if this process has not been used more than five times"), or financial constraints (e.g., "execute only if the specified account received a payment"). Thus, they can represent arbitrary rules and constraints, turning the system into a flexible environment for compositional logic.

All three types of components are recorded on the blockchain with unique identities, allowing others to reuse them, build on them, and reference them in their own processes. These building blocks remain inert until activated as part of a Performative Transaction, described in the next section.

Performative Transactions

A Performative Transaction (PT) is the fundamental unit of execution within the DCN. It represents the moment when a compositional process—defined through features,

transformations, and conditions—is activated. This activation can generate outputs, trigger other transactions, and update the state of the system, depending on how the PT is designed. The term "performative" is used here to indicate that the transaction performs a process: it executes compositional logic and produces an outcome.

Each PT consists of:

1) a named feature with its list of transformations,
2) an optional list of dependent PTs,
3) a condition that must be satisfied for its execution.

For example, a user might create a PT that generates a melody whenever called (condition: always true). Another user could then build a new PT that combines that melody together with another rhythmic pattern as a dependent PT and applies another PT that inverts the intervals, but only when the weather is good. When a user or agent executes the PT via the API, they supply:

- N—how many values they want generated,
- a startValue for each referenced PT—indicating which transformation in its sequence to begin from.

The system resolves all references recursively, down to the lowest-level components, applying the transformations and checking that conditions are met. This produces an output array of scalar index values—which can be interpreted as pitches, durations, dynamics, links to external media, or any other digital parameters, depending on how the user maps them in their application. These outputs can be rendered as music or notation, used as performance cues, or even fed back as inputs for further PTs (as discussed further in "What Are We Modelling? Indexes, Structures, and Interpretation").

What distinguishes a PT from a generic function call is its compositional and referential capacity. When you reference another user's PT in your own PT, you are not copying it—you are executing that original component live, with its own logic, authorship, and constraints. This introduces several important qualities:

1) Traceability—every PT retains the provenance of its components; you know exactly which PTs were used, in what order, and by whom.
2) Modularity—PTs can be built on top of one another, resulting in complex,

layered compositional logic.

3) Interoperability—because each component is defined and executed through a standardised API, any PT can interact with others, regardless of who authored them or how they are used.

The public API enables several modes of use:

1) direct use by humans via a User Interface or Command Line Interface (e.g., composing with reusable features),
2) indirect use through AI agents, which can generate, recombine, or post new contributions (e.g., prompting a Large Language Model to create a transformation, feature, or condition),
3) recursive use within the system itself, where a transformation can trigger an API call—either to the DCN itself, or to external AI agents—allowing the system to restructure itself as part of its own operations. This means that the architecture enables the mode of allagmatic composition discussed in Chapter 4: the autopoietic system not only generates content but also participates in reorganising its own logic through PTs.

A PT is thus the crystallisation of a creative moment—a decision to activate a particular compositional structure with specific parameters. This is what allows the DCN to function not just as a storage layer or generative engine, but as a living system of transactions—a recursive, collaborative, programmable space for artistic composition.

Composing and Testing Transactions

While PTs are ultimately executed on the blockchain, the process of composing, testing, and refining them does not require direct interaction with the distributed blockchain infrastructure. To make the system accessible and practical for users, the DCN server[8] functions both as an interface and a test development environment. It acts as an intermediary between users and the blockchain, exposing the public API and simulating the execution of transactions in a local blockchain without incurring

[8] The open-source implementation of the server is hosted and can be accessed under the URL address: https://decentralised.art.

any fees for its users before modifying the distributed blockchain's state.

For users, this means that the creative process can unfold fluidly. You can define a PT, try out different PTs, test how they behave over a range of inputs, and explore how they interact—all in a simulated environment. When you are satisfied with the result, you can choose to publish the PT to the distributed blockchain, where it becomes part of the shared compositional ecosystem. Only then, the decentralised network will require a gas fee for including the new transaction on-chain.

This development environment offers several key advantages:

1) Costless experimentation: Because blockchain transactions typically involve fees, composing directly on-chain would be prohibitively expensive for exploratory work. The server simulates the Ethereum Virtual Machine (EVM), allowing users to test PTs as if they were live, but without any financial cost. This makes the system practical for creative iteration and improvisation.
2) Immediate feedback: Users can experiment with different parameter values—such as N (the number of outputs to generate) or startValue (the index from which generation begins)—and see how these affect the output of a given feature. This is particularly important when working with complex, layered features or when reusing processes created by others.
3) Accessible for non-programmers: while the low-level logic of PTs is implemented in Solidity programming language, users do not need to write blockchain code to use or recombine existing components. Many will interact with the system through graphical interfaces, CLI tools, or application-specific UIs built on top of the API. These interfaces rely on the server as a stable and programmable backend.
4) Open-source and self-hosted options: the server implementation is open-source and publicly available. Users can access the public instance and interact with the live network. However, users and developers who prefer not to rely on shared infrastructure can install and run the server locally, giving them full autonomy and control over their interaction with the blockchain network.
5) Programmable by humans and agents alike: the API exposed by the server can be used not only by human users but also by AI agents and autonomous scripts. For example, an AI agent could use the API to query available features, construct a new transformation chain, execute a test PT, and post the result—entirely without human intervention. This design opens the door for agent-

based co-creation, where human composers, algorithmic systems, and AI agents all participate in building and expanding the network.

The result is a layered architecture: while the blockchain guarantees persistence, traceability, and authorship, the server ensures usability, flexibility, and freedom to experiment. For artists, this means that composing with PTs becomes a live, iterative process. For developers and researchers, it means that new tools, agents, or applications can be built directly on top of the infrastructure without modifying its core. This separation of execution environments—on-chain for publication and off-chain for development—makes the DCN not just a compositional protocol, but a complete environment for creative experimentation and distributed authorship.

What Are We Modelling? Indexes, Structures, and Interpretation

The DCN does not produce music, images, or performances directly. Instead, it generates and manipulates indexes—integer numbers that acquire meaning only when interpreted by a user, application, or agent. This is a crucial shift in perspective: what is encoded on-chain are not artworks or media files, but compositional structures—systems of operations that unfold according to defined logic and yield numerical outputs. These outputs can then be mapped to musical pitches, durations, dynamics, gestures, samples, files or any other symbolic or material domain, depending on the context in which they are used.

At the lowest level, the outputs of a PT are arrays of scalar indexes. These indexes are not musical notes, events, or sounds. They are integers that serve as placeholders or coordinates within a potential space of interpretation. Their meaning emerges only when mapped, either in software or performance, to a particular set of symbols or parameters. An index such as "7" (counting from 0) might refer to the eighth pitch in a chromatic scale, the eighth rhythmic subdivision in a phrase, or the eighth image in a visual sequence. To be clear, it doesn't have to map to anything "eighth" in order at all. It can map to anything, which just means that the eighth element in a list is that mapped thing—a file, a numerical value, a graphic, etc. The index itself is agnostic; the user or the rendering agent defines what it signifies.

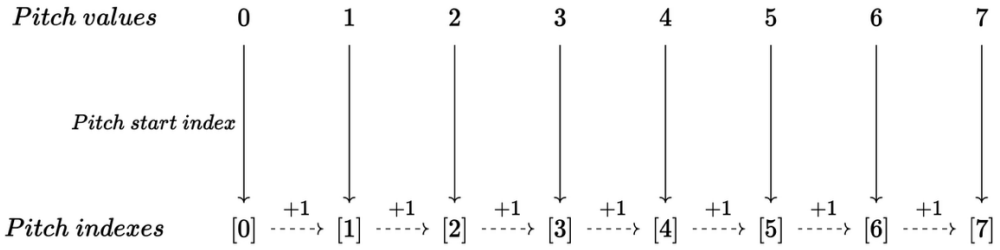
For that reason, at the heart of every PT lies a "scalar feature", which is a list of simple integer numbers (0, 1, 2, 3, 4, 5 … etc) that can be further mapped to any list of digital

objects or compositional logics. Such a scalar feature should be best named based on what type of objects or processes it is mapped onto (e.g., a scalar feature of integer numbers from 0 to 127 could signify MIDI notes, or a scalar feature of integer numbers from 0 to 15 could signify available rhythmical values to be chosen from in the composition).

Suppose we define a scalar feature Pitch.

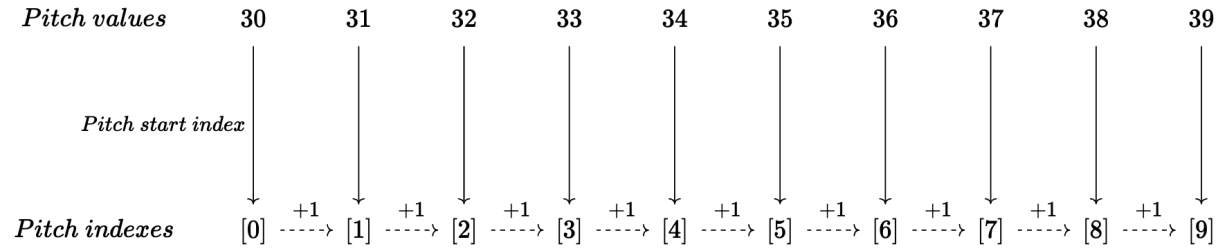
A scalar feature is a reusable generator of indexes. On every step the feature adds 1 to the previous index. Once a scalar feature is defined, it can be run with any integer number for its starting index and for its N value, which is a number of values to generate.

With a start index = 0 and N = 7, the run unfolds as:



returning the array [0, 1, 2, 3, 4, 5, 6, 7].

With a start index of 30 and N = 10, the run unfolds as:



returning the array [30, 31, 32, 33, 34, 35, 36 ,37, 38, 39].

The feature does not know it's generating "pitches" (values)—it only knows it's applying a transformation "addOne" over an index space. Scalar features should be thought of as collections of all possible indexes for one of the dimensions that the composition can be built of, such as all of the possible frequencies in an audio file, all

possible pitches in a score, all possible rhythms, samples, patterns, etc. For each "type" of such a dimension, we define a separate scalar feature.

Once scalar features are defined, we can compose by selecting various combinations of indexes from them and by combining them with each other. To do it, we use transformations.

Transformation is a programmable function that maps one index to another. As W. Ross Ashby defines it in "An Introduction to Cybernetics": "a set of transitions, on a set of operands, is a transformation", and it is "concerned with what happens, not with why it happens" (Ashby, 1956, 10–11). Ashby identifies difference as the most fundamental concept in cybernetics: "either that two things are recognisably different or that one thing has changed with time" (9). Transformations in this system are thus formal mechanisms of change—units that produce difference without requiring interpretation. This system assumes all such differences occur in finite steps—"by a measurable jump" (9)—aligning with the discrete, index-based logic of PTs.

As already mentioned, scalar features always use the same type of transformation:

```
function addOne(x) => x + 1
```

This transformation could be interpreted as ascending steps in pitch, time, spatial position, or other dimensions. However, we can further define various other transformations that instead of generating the "next" index, they "select" indexes in a changed order. For example, a transformation that selects every second index:

```
function addTwo(x) => x + 2.
```

The logic of transformation can contain any executable computer code, but the main rule is that the transformation takes one index as its input and returns one index as its output. This enables a feature to use transformations for generating and accumulating indexes when it is run with various parameters.

Various transformations can be used to build intricate compositional patterns. To do it, we define a "composite feature" (in a PT that uses other PTs as dependencies), which uses features of other PTs as a source of elements and performs transformations on top of them to acquire a selection of values. The user chooses the PT on top of which the selection will be made and composes a chain of transformations that will be performed on top of that PT's feature, to select it's indexes.

Let's explore how this works in detail. Suppose that the previously-defined scalar feature Pitch is mapped onto MIDI notes corresponding to musical pitches, where index 60 is the middle C.

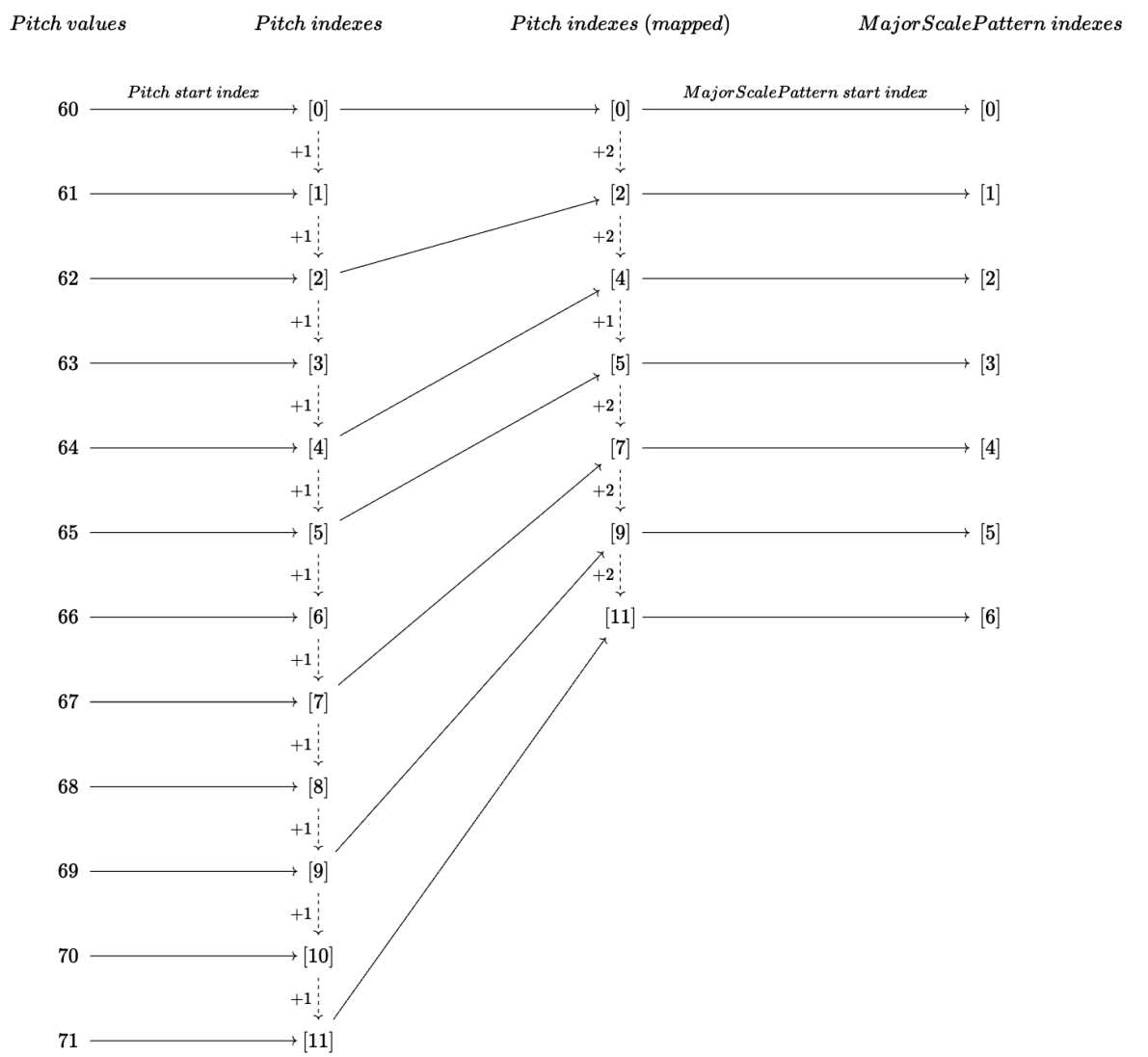
We define a composite feature MajorScalePattern. We choose the previously defined "Pitch" scalar feature as its sub-feature. We also chose the previously defined transformations "addOne" and "addTwo" in order to select a configuration of indexes that would correspond to MIDI notes shaped into a Major Scale pattern:

```
Feature: MajorScalePattern

Sub-feature: Pitch

Transformations: [0: addTwo, 1: addTwo, 2: addOne, 3: addTwo,
4: addTwo, 5: addTwo, 6: addOne]
```

Once such feature is defined, we can run it to obtain a result. To do it, the user should provide:

1) A start value for the Pitch sub-feature (from which index should it start generating further indexes)—in the following example, let's try 60.
2) A start value for the MajorScalePattern (from which index of the subfeature should it start applying its transformations to obtain indexes) —in the following example let's try index 0 (index value of the subfeature Pitch).
3) A transformation index of the MajorScalePattern (from which transformation in a defined chain of transformations should it start selecting the indexes iteratively)—in the following example let's start from 0, which is the "addTwo" transformation (see the definition of the MajorScalePattern feature above).
4) A global N value for deciding how many samples to generate using the feature—in the following example let's try 7 indexes.

When the feature MajorScalePattern is run with the chosen parameters, it applies its list of transformations on top of the index space generated by its subfeature Pitch:



Important: observe that the MajorScalePattern's transformations such as addTwo and addOne modify the indexes of Pitch, NOT the Pitch values under these indexes. The transformations are performed on indexes, not on the values that they represent.

To provide another example, now let's first define a scalar feature Duration, assume that its index [1] symbolises a duration of a sixteenth note, and that each next index symbolises a doubling of that value (eighth note, quarter note, half note, whole note, etc.):

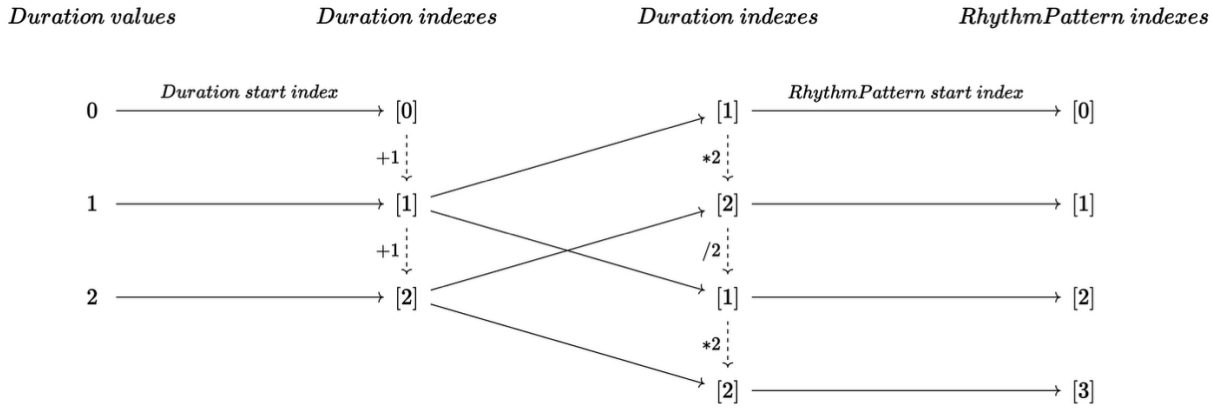
```
Scalar Feature:  Duration
```

Further, we define a composite feature RhythmPattern that applies a pattern of transformations on top of the Duration's scalar feature:

```
Feature:  RhythmPattern

Sub-feature: Duration

Transformations: [0: multiplyByTwo, 1: divideByTwo]
```

To run this PT, we need to provide a start index for the dependent PT Duration (let's try 0), a start index for the PT RhythmPattern (let's try 1), an index of transformation to start generating with (let's start with 0), an a global N value for the number of samples to generate (let's generate a rhythm of 4 values).

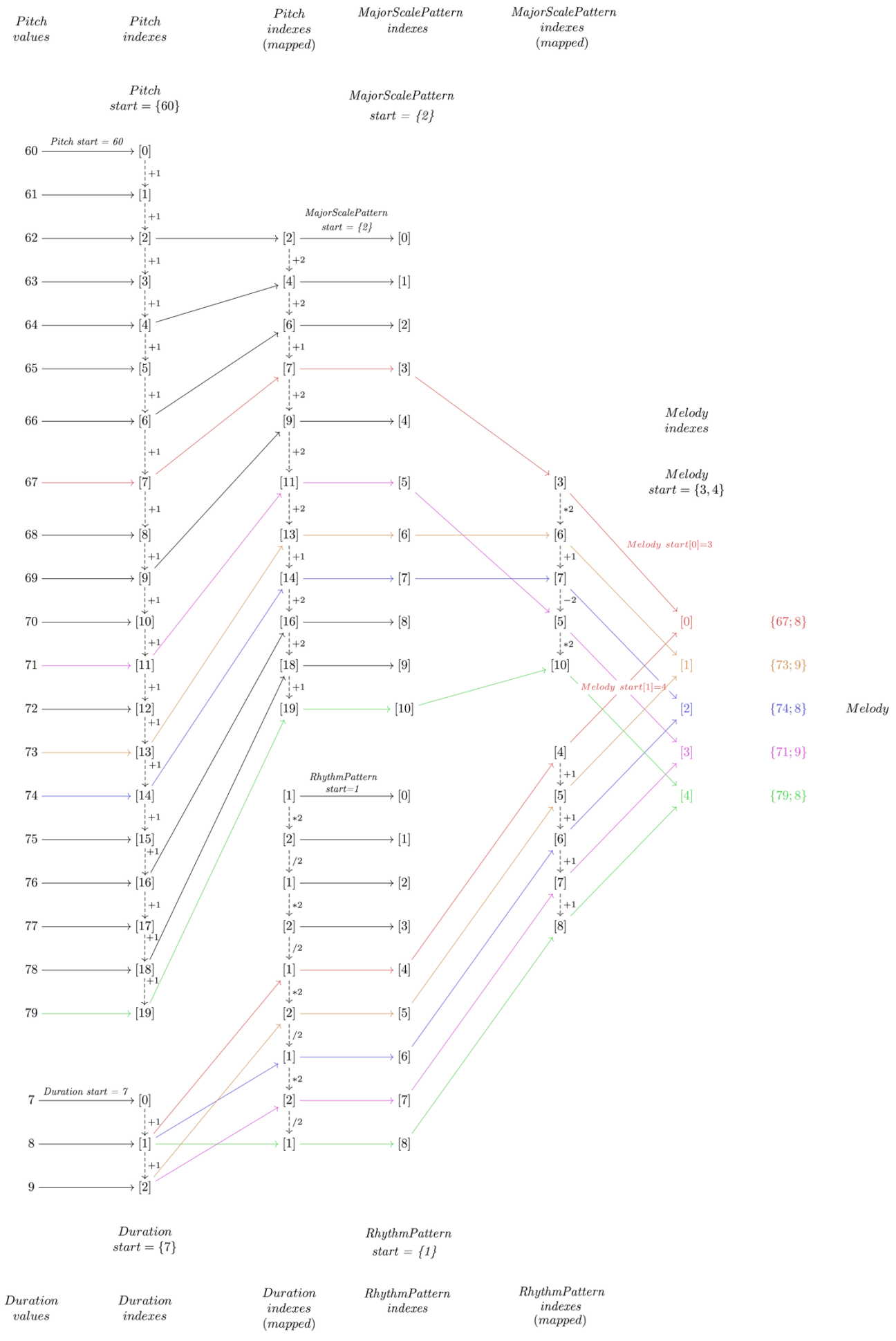
The PT runs as follows:



The result is the array of N=4 values: [1, 2, 1, 2].

As we have alread defined two scalar features (Pitch and Duration), and two composite features (MajorScalePattern and RhythmPattern), we could now combine them further to create a melody:

```
Feature: Melody

Sub-features:

[0: MajorScalePattern, 1: RhythmPattern]

Transformations: For MajorScalePattern: [0: multiplyByTwo, 1:
addOne, 2: minusTwo]For RhythmPattern: [0: addOne]
```


Pitch values
Pitch indexes
Pitch indexes (mapped)
MajorScalePattern indexes
MajorScalePattern indexes (mapped)
Pitch start = {60}
MajorScalePattern start = {2}
Pitch start = 60
MajorScalePattern start = {2}
Melody indexes
Melody start = {3, 4}
Melody start[0]=3
Melody start[1]=4
RhythmPattern start=1
Duration start = 7
Duration start = {7}
RhythmPattern start = {1}
Melody
{67; 8}
{73; 9}
{74; 8}
{71; 9}
{79; 8}
Duration values
Duration indexes
Duration indexes (mapped)
RhythmPattern indexes
RhythmPattern indexes (mapped)

The system handles this automatically: it calls the sub-features, applies their transformations, checks any conditions, and outputs final scalar indexes.

To run the Melody PT, start indexes have to be provided for each used feature and sub-feature. Executing Melody results in a sequence of Pitch and Duration values. As traceable on the figure, this PT first runs its scalar features to calculate their indexes. Further, MajorScalePattern and RhythmPattern run to select their indexes. Finally, Melody selects its final indexes to form a new melody based on pitch and rhythm.

From the system's point of view, Melody contains a composite feature with two branches. While the operations presented on the diagram here might seem complicated and difficult to follow, this is exactly where the biggest strength of the system lies. The system handles all of these calculations automatically: it calls the sub-features, applies their transformations, checks any conditions, and outputs the final values. The user doesn't need to think in terms of iterations and complicated trees. They simply combine pre-existing features and execute them to test the final outputs.

Both scalar and composite features can be further used as building blocks within other PTs. In this way, the PT called Melody, although itself a composite structure, could be nested inside larger compositional constructs—such as phrases, motifs, or full scores—without any change to its internal logic. For example, one could define a new PT VariationA that references Melody as a dependent PT and applies further transformations to its output. Another PT, VariationB, might use the same dependent PTs as Melody but change the transformation pattern, or modify the starting index and N values to produce a different walk through the same underlying space.

This iterative architecture allows any PT to serve as a self-contained, reusable module within a broader system. By chaining PTs across multiple levels, users can construct highly structured generative processes using a minimal set of operations. Each layer encapsulates its logic, and higher-order features manipulate only the index space exposed by their subfeatures—never needing to understand their internal workings. This principle of compositional encapsulation enables a powerful mode of structured abstraction, where creative variation emerges by recombining, reparameterising, and repurposing existing components.

While many transformations are deterministic—returning the same output for a given input—it is also possible to define non-deterministic or stochastic transformations. For instance, a transformation might introduce randomness by returning a different value

each time it is called, even with the same input. A transformation such as RandomWithinRange(min, max) could be programmed to return a pseudo-random number between min and max. This introduces controlled variability into the system, enabling the generation of different outcomes from the same structural logic. Such stochastic processes can be particularly valuable in generative music, where variation and unpredictability are key compositional resources.

Just as transformations introduce difference through operational logic, conditions introduce difference through contextual logic. They don't enforce limitations so much as define the creative circumstances under which a feature may unfold. Conditions allow features to respond to their environment—to be activated only under certain temporal, social, or physical configurations—enabling new forms of situated, responsive composition. Conditions can be illustrated simply. For example, a reusable condition called MaxFivePlays:

```
Condition: MaxFivePlays
```

```
Logic: Allow execution only if the feature has been triggered < 5 times
```

This would block a PT after its fifth use.

Other conditions might involve access control or interaction with external data. A condition named TokenHolderOnly could restrict execution to users who hold a specific token in their wallet, effectively tying compositional access to ownership within a broader ecosystem. More complex conditions can query off-chain information through blockchain oracles—mechanisms that securely feed real-world data into smart contracts. For example, a condition called IfSunnyInWarsaw could be programmed to execute only if a trusted weather oracle reports clear skies in Warsaw. Another condition, MinAudience10, might monitor the number of active viewers on a platform and allow execution only if at least ten users are watching the artwork simultaneously—creating a form of social activation. Finally, sensor-integrated conditions are also possible. A condition like HumidityBelow60 could be connected to a room-based sensor, executing the underlying feature only when local humidity levels fall below a defined threshold. These examples show how conditions can embed compositional logic within wider artistic contexts, transforming the artwork into a responsive and situated system.

What the infrastructure models, then, is not the artwork itself but the logic that generates its constituent parts. The system allows users to define how one value leads to another, how values relate within and across dimensions, and how those relations can be composed into more complex operations. Each PT defines a structure: a series of transformations that recursively unfold to produce a stream of indexes. These structures may be simple (such as a linear sequence using a single transformation) or highly composite, consisting of nested features that apply different operations in parallel or in sequence, each governed by its own logic and conditions.

From the perspective of the system, there is a key difference between scalar features and composite features as explained before. A scalar feature is terminal: it generates a sequence of raw indexes and does not depend on any other features. It represents a single operational line. A composite feature, by contrast, is defined in terms of other features. It selects and transforms the outputs of its sub-features, assembling them into higher-order structures. Conceptually, one can think of composite features as defining operations on operations—processes that organise other processes. But from the user's perspective, these distinctions may be invisible. Whether a feature is scalar or composite has no bearing on how it is used or executed. All features can be executed in the same way—by calling them with a given N and startValue—and all produce scalar outputs that can be interpreted downstream.

What matters is that composite features introduce a new kind of modelling: not of values themselves, but on indexes that signify relations between values. When a user creates a PT with a composite feature, they are effectively constructing a graph—a directed structure in which each node is a PT and each edge represents a transformation. The system executes this graph recursively, starting from the top-level PT and resolving all dependencies until it reaches the terminal scalars. This graph is not hard-coded; it is assembled dynamically at the moment of execution, based on the features and transformations referenced in the transaction. This means that every execution is both a computation and a traversal of a user-defined compositional topology. Every node can be used to select values from (or "metamodel") any other node in the network.

The power of this metamodelling approach lies in its generality. Because scalar outputs are only meaningful when interpreted externally, the same infrastructure can support an enormous range of applications: music, choreography, image generation, text sequencing, algorithmic theatre, or even non-artistic domains. What is shared

across all these cases is the logic of indexed generation, transformation, and combination. The system models not the end products, but the operations that give rise to them. In this sense, it is a metamodel: a system for composing compositional processes, and for recursively modifying their operational relations.

What the DCN models is not musical content, but the operational structures through which such content can be generated. It encodes relations, not representations. And it does so in a way that is open-ended, iterative, and designed to support a plurality of interpretations (the same node can be reused in various artistic contexts). This is what makes it not just a tool for composition, but a compositional system in itself—one that can be shaped, expanded, and recomposed through the very transactions that take place within it.

This orientation toward operational logic, rather than fixed content, invites a reconsideration of what constitutes a musical "work". If outputs are endlessly variable, yet structurally derived from a stable procedural form, what defines the identity of a composition? As Edward Campbell observes, Harrison Birtwistle once claimed he "could have composed the same piece with entirely different pitches", raising the question of compositional identity as distinct from sonic material (Campbell, 2013, 34). Mauricio Kagel went further, composing Morceau de concours twice—in 1971 and again in 1992—with the two versions sharing "not a single note in common," yet still understood as iterations of the same piece (Heile 2006, cited in Campbell, 2013, 34). The DCN operates in a similar conceptual space: it is the space of operations and links between them that persist.

How the Infrastructure Works

Behind the user-facing simplicity of features, transformations, and conditions lies a carefully structured technical architecture. It ensures that compositional processes are executed correctly, persistently, and traceably on the blockchain. While users interact with the system through an intuitive API—building and recombining processes as artistic components—the underlying infrastructure handles registration, execution, state updates, and provenance tracking in a decentralised, modular, and verifiable way.

At the heart of this system is a set of Ethereum smart contracts. These contracts define the formal logic of PTs and the objects they manipulate. When a user defines a new transformation, feature, or condition, they are essentially creating a new contract that adheres to a shared interface. The logic of a new transformation is written in Solidity—the programming language for Ethereum smart contracts—but the broader blockchain interactions, such as how this transformation will be executed, referenced, or composed, are abstracted away. This separation allows users to focus on writing the transformation logic itself without having to manually define low-level blockchain interactions. In this sense, the infrastructure offers a structured but open-ended programming model: the rules of composition are strict enough to guarantee interoperability and execution, yet open enough to accommodate any kind of transformation logic.

Each component that users contribute is registered on-chain via a central registry contract. This registry acts as a decentralised index, associating unique human-readable names with their corresponding smart contract addresses. The registry ensures that all components are discoverable and verifiable, enabling any user to reference components authored by others. When a user creates a new PT, they can declare which existing components it builds upon. These declarations are stored as part of the feature's metadata and can be resolved programmatically during execution. Importantly, this also guarantees traceability: every compositional object carries within it a complete record of the components it depends on and the order in which they are invoked.

The actual execution of a PT is coordinated through another component of the infrastructure: runner. The runner is a smart contract responsible for initiating the execution of features, resolving their dependencies, applying transformations, and checking whether conditions are met. When a user sends an execution command—specifying a feature name, an index value, and the number of values to generate—it is the runner that traverses the feature structure, recursively applies all transformations, and produces the output sequence. During this process, the runner interacts dynamically with the registry to retrieve the correct components, and with each component's contract to execute its specific logic. This recursive traversal ensures that even highly nested, multi-layered features can be executed in a deterministic and reproducible manner.

The runner performs this work on-chain when a PT is executed in the blockchain environment, but the same traversal logic is replicated off-chain in the development server. This duplication ensures consistency: the results of off-chain testing correspond exactly to the results of an on-chain transaction. Users thus gain the benefits of both worlds—freeform, costless experimentation in the development environment and verifiable execution and authorship in the decentralised network.

While the registry and runner provide the scaffolding for execution, the server layer—described earlier—provides the necessary bridge between users and this compositional infrastructure. It exposes the API endpoints that allow users to query registered components, simulate executions, test conditions, and post new features or transformations. Developers building user interfaces, AI agents, or other software tools can rely on this API as a stable, predictable interface. In turn, the server communicates with the blockchain itself, abstracting away the complexities of transaction signing, gas estimation, and error handling.

Through this architecture, the infrastructure enables users to treat compositional logic not as a static score or encoded object, but as a living, executable process. Each new contribution enters a shared compositional space where it can be accessed, invoked, or recombined by others. The registry guarantees discoverability, the runner guarantees correctness of execution, and the blockchain guarantees persistence and authorship. At the same time, the off-chain server layer allows creative experimentation to remain open-ended, iterative, and responsive.

This division of roles—between blockchain and server, between transformation logic and system execution—ensures that the DCN remains simultaneously robust and generative. It does not dictate how users should compose, only how compositional processes are stored, connected, and activated. What emerges is a protocol for building compositional ecosystems.

AI Agents and Recursive Infrastructure

One of the defining features of the DCN is that it is not only usable by humans, but also designed to be interoperable with software agents—particularly AI systems. This opens a new domain of compositional practice, where agents can participate as contributors, users, and even meta-composers within the system. Crucially, these

agents are not peripheral add-ons. They can operate both externally to the network, using the API like any human would, and internally, embedded within transformations that themselves call the network during execution. This dual capacity allows AI agents to act as recursive infrastructure-builders within a live compositional system.

One of the most promising developments in this area is the capacity of Large Language Models (LLMs) to operate as reasoning agents capable of executing code and interacting with APIs (Eleti et al., 2023; Yao et al., 2023; highlighted in Łukawski, 2024, 45). This functionality allows LLMs not only to query and compose with existing features on the network, but also to generate new ones, manage complex logic chains, and post PTs autonomously. In this model, LLMs can curate and build evolving collections of artistic processes—collections which, once posted, become available to other human and posthuman agents for further recombination. LLMs can also serve as natural language interfaces between users and the network, translating intuitive prompts into executable PTs (Łukawski, 2024, 45). This deepens the recursive entanglement between agents and infrastructure, where AI is not just an assistant but a fully embedded co-actor shaping the evolving space of composition.

To clarify how this works in practice, let us consider three distinct use cases.

1. External AI agent using the API like a human user

An AI agent can be programmed to browse the network via the public API, query for existing features, and construct new features by combining them. For example, a generative agent might:

- Search the network for all features tagged with "rhythm" and "duration",
- Select one at random or by a heuristic (e.g., sparsity, entropy),
- Combine it with a pitch-generating feature using a transformation such as Add or Invert,
- Wrap the result in a new feature and post it back to the network.

This results in a new PT authored by an AI agent. The feature may be executed by a human composer later, or may be picked up by another agent in future runs. In this

way, the AI behaves like any other network participant—exploring, composing, and contributing via the API.

2. A transformation that calls an AI model from the inside

More complex behaviours emerge when AI agents are embedded directly in transformations. Consider a transformation named MLHarmonySelector. Its logic might be:

```
function run(x) returns (uint32) {
  // send input index x to an external AI model via API call using blockchain oracle
  // receive a predicted harmony index y based on a trained style model
  // return y as the transformed value
}
```

Here, x might be the current position in a phrase. The transformation sends it to a machine learning model (e.g., a Transformer trained on chorales), which responds with the most likely harmonically appropriate value y. This value is returned as the output of the transformation and continues through the feature's execution.

From the system's point of view, nothing special happened: it simply called a transformation. But that transformation reached outside the chain of purely algorithmic operations, consulted an AI, and folded its result back into the compositional process. This enables non-deterministic, data-driven, stylistically-informed decision-making within the otherwise deterministic logic of PT execution.

3. Recursive AI-based transformation that modifies the network

The most advanced form of agency occurs when a transformation uses the API during execution to modify the compositional infrastructure itself. Imagine a transformation called AutopoieticAgent that, in response to a given index, does the following:

- Queries the API to retrieve all features tagged "melodic-pattern".
- Evaluates them using an internal scoring function (e.g., novelty, diversity).
- Generates a new variation using an AI model.
- Posts this variation to the network as a new feature.
- Returns the index of that new feature as the transformation result.

This is a PT that not only transforms a number, but expands the space of possible transformations for future users. It effectively composes new compositional logic.

Because the infrastructure is recursive and open-ended, this type of agent-based transformation is not speculative. It is directly enabled by the architecture. The same API that a human uses to post a new feature can be called from within the transformation itself. This allows PTs not to be just compositions, but meta-compositional events—modifying the system as they run.

These three examples illustrate how AI agents can operate at multiple layers of the Decentralised Creative Network:

- as external users,
- as embedded transformation logic,
- and as recursive participants that shape the network from within.

The boundary between composer, performer, and system is blurred. A PT may originate from a human, but contain transformations authored by AI, which in turn call other agents to generate outputs or spawn new features. This deep entanglement of agents and infrastructure is a direct consequence of exposing the compositional system as a public API.

In this model, AI is a co-actor in the compositional ecology, capable of contributing both content and infrastructure. What emerges is a new kind of compositional logic: distributed and autopoietic—where agency lies in the continuous shaping of the network that makes composition possible.

Case Studies: *Ani(mate)* (Movement VI), *Allagma*, and *Chain of Thoughts*

1. Ani(mate) for piano and strings (Movement VI):

In Chapter 1, I introduced *Ani(mate)* as a commissioned work whose first five movements were composed through connectionist, curatorially driven methods. Having now established Performative Transactions and Decentralised Creative Networks as the core operational constructs, I return to *Ani(mate)* and extend the discussion to two further works developed during this doctoral trajectory.

The purpose of this section is to show, in practice, how PTs and the DCN function as a compositional infrastructure and how they reframe authorship, identity, and provenance in three complementary case studies: the sixth movement of *Ani(mate)* (a PT-composed movement selected for a fixed score), *Allagma* (an agentic two-phase composer for any MIDI instrument), and *Chain of Thoughts* (an ensemble piece generated from a score of prompts). Together, these works can be read as a progression from human-defined vocabularies, through agent-generated vocabularies, to composer-specified prompt chains that the agent realises within the DCN.

The sixth movement of *Ani(mate)* was composed as the first full-scale artistic experiment with the test version of the DCN, created entirely from compositional structures authored as PTs. The aim was to investigate whether a self-contained vocabulary of modular generators—defined through features, transformations, and conditions—could produce musical material whose identity remained traceable through the network's provenance structure.

I began by defining a small set of permissible transformations (limited to additive and subtractive operations on integer streams) and applying them to terminal scalar features such as pitch, time, duration, and velocity. These scalar features were then combined into higher-level composite features, which functioned as modular "generators". Large Language Models were used here as co-authors at the design stage: at each stage I used the model to propose new candidate musical features, which I then reviewed and refined before registering them on-chain. The resulting vocabulary formed a matrix of interdependent generators whose relations were explicitly recorded in the DCN registry.

To generate the musical material, I executed these generators as PTs with varying

sample counts (*N*) and starting seeds. This produced families of structurally related yet distinct outputs, all of which retained their provenance links to the originating features and transformations. From this collection, I curated a selection of outputs to notate and orchestrate as the final score of the movement.

Working with the DCN in this way influenced the compositional experience in several important respects. Unlike the connectionist workflow used in movements I–V, composing this movement felt highly controlled. Each generator was constructed as an explicit configuration of transformations applied to musical features, designed and adjusted in a symbolic, rule-based manner. Because these transformations operated on predictable integer streams, I could reason about their effects before execution and verify them immediately using the MIDI player. This approach enabled precise interventions in the internal logic of the composition while maintaining a coherent structure across its outputs.

The provenance structure of the DCN also had a strong impact on how I listened to and evaluated the material. Because every generated fragment could be traced back to the exact configuration of features from which it emerged, it became possible to read the music as a graph of decisions. Every fragment I generated could be traced back to the exact constellation of features and transformations from which it emerged, making the relationship between cause and effect unusually transparent. This shifted my evaluative focus: rather than judging fragments as isolated products, I could understand them as outcomes of specific procedural definitions, and adjust those definitions accordingly.

2. *Allagma* for MIDI instruments*:*

While *Ani(mate) VI* explored the manual construction of a compositional vocabulary within the DCN, *Allagma* extends this approach by delegating the creation of that vocabulary to an autonomous agent. The work is conceived for any MIDI-controlled instrument and will be premiered during the final PhD concert at Studio 6, Amare (Den Haag) on the 20th of November 2025, performed using instruments including a piano voorsetzer (which enables an acoustic grand piano to be played via MIDI) and the microtonal MIDI organ robot *4Pi*, both constructed by composer, inventor and instrument-builder Godfried-Willem Raes and delivered by the Logos Foundation.

The piece is composed as a self-contained composer: a program that generates a new realisation each time it is run. It operates in two distinct phases. In Phase 1, the system prompts a Large Language Model to design a set of PT features that will serve as its own vocabulary. These features are generated and registered on the DCN as reusable compositional modules. The agent evaluates its own previous outputs during this phase, using the evolving vocabulary as context for producing further features until a diverse collection has been assembled.

In Phase 2, this vocabulary is used to generate the composition itself. Each vocabulary element is executed multiple times as a PT, producing streams of musical data (MIDI pitch, time, duration, velocity, etc.) from different seeds and transformation offsets. These streams are then passed back to the Large Language Model, which receives summaries of their temporal spans and is instructed to act as a scheduler, determining when each generated fragment should begin within the overall timeline of the composition. This ensures that the resulting piece is not a simple concatenation of fragments but a structured form with interleaved episodes, reprises, and silences. The final output is exported as a unified sequence of MIDI data for performance on the robotic instruments.

The process used for the generation of the work's iterations embodies the principle of allagmatic composition introduced earlier: the work exists as as a system that continually reconfigures its operational structure to produce new versions of itself. Each execution of *Allagma* represents a distinct individuation of the same underlying system, preserving the procedural identity of the work while producing divergent musical outcomes.

Compared to *Ani(mate) VI*, which relied on a manually constructed and tightly supervised vocabulary, *Allagma* shifted the compositional act from designing the vocabulary itself to designing the conditions under which a vocabulary could emerge. This reframed my role: rather than specifying individual transformations, I specified the meta-level rules and evaluation criteria through which the agent could construct its own set of compositional primitives. The resulting material often contained structural relationships and textural ideas that I would not have devised myself, yet these were generated within a framework that preserved traceability through the DCN's provenance graph. This made it possible to analyse and iteratively refine the agent's evolving vocabulary while allowing its internal logic to develop autonomously.

Crucially, this experiment demonstrated that allagmatic composition does not require human authorship at every level of the system. What matters is not who defines each element, but that the system preserves and exposes the operational provenance linking those elements together. In *Allagma*, the human-computer relation became one of co-construction rather than curation: I built the operational boundaries and evaluation criteria, while the agent populated that space with compositional content. Each new iteration of the piece thus acts as a record of the agent's reasoning process within a set of constraints, revealing how meaning and structure can arise from the recursive interplay between autonomous and human-specified contributions.

3. *Chain of Thoughts* for ensemble:

The third case study in this sequence explores a compositional model built not from pre-defined vocabularies but from *chains of prompts*. Instead of specifying musical material directly, I wrote structured textual instructions that guide an AI agent to compose one bar at a time within the DCN. Each instruction is interpreted in the context of everything the agent has generated so far, allowing it to shape the ongoing musical discourse through a continuous process of contextual reasoning.

A typical instruction might read as follows:

"Compose the next bar for alto flute and violin. Use material that contrasts the previous bar's staccato texture with sustained tones. Reuse one pitch from the previous bar's pitch collection, but place it in a different register. Maintain a dynamic level around mezzo-piano. Ensure rhythmic alignment between the two instruments, with note onsets every quarter beat".

When receiving such a prompt, the agent queries the DCN for relevant existing features (e.g. pitch sets, rhythmic cells, articulation patterns), generates new features if needed, and resolves the instruction into concrete data streams for pitch, time, duration, and velocity. These outputs are registered in the network as new PTs and immediately incorporated into the developing score. Each new bar thus depends not only on its local instruction but also on the accumulated material history of the composition, which is passed to the agent as context at every step.

This system forms the basis of the composition *Chain of Thoughts*, written for Ensemble Modelo62 and scored for alto flute, violin, bass clarinet, trumpet, cello, and double bass, to be premiered at Studio 6, Amare (Den Haag) on the 20th of November 2025

under the direction of conductor Ezequiel Menalled. The piece exists not as a fixed musical text but as a score of interdependent prompts: each run of the system realises these prompts as a new version of the work, while preserving the provenance of their underlying operations in the DCN.

Unlike *Allagma*, which relied on an AI agent to autonomously construct its own vocabulary before composing with it, this piece involved writing a fixed score of agential operations specified as a chain of prompts. Each bar of the composition is defined by a detailed natural-language instruction describing how its musical parameters should be generated, but without prescribing exact notes. These prompts give the agent controlled ambiguity: enough freedom to produce locally varied material, yet constrained by the precise operational role assigned to each bar. For the piece to be performed, first the AI agent (or actually human agent for that matter) has to execute this sequence of prompts, accessing and creating features in the DCN as needed, and rendering the result as musical notation. Every run of the system thus produces a new score, yet all realisations share the same internal structure of compositional operations encoded in the fixed chain of prompts.

Building on the Infrastructure: Interfaces, Applications, and Experimental Economies

As the technical architecture of the DCN becomes operational, the next step is to design diverse interfaces and applications that make its compositional logic accessible, legible, and usable. Without well-considered user environments, even the most powerful systems remain under-utilised. As Madeline Gannon aptly notes, reflecting on the public impact of ChatGPT: "Interface is Everything. (…) Invest the time to make it easy-to-use and easy-to-understand by the general public … and lo and behold! It takes the world by storm" (Gannon in Cuan, 2023, 353). This observation is particularly salient in the case of DCNs, where the underlying system is intentionally open-ended but may appear opaque to non-technical users.

What is needed now is a proliferation of decentralised applications (dApps) that engage with the public API and offer tailored tools for creative practice. These applications might include: interfaces that allow users to define, test, and share their own low-level transformation functions without needing to write Solidity code, visual

environments for composing features by linking existing components, inspecting dependencies, and defining transformation chains, tools for crafting and combining conditions based on state, oracles, or sensor data—enabling complex and responsive creative logic, playback engines, notation renderers, or spatial mappers that interpret scalar indexes in musical, visual, or performative terms, systems for sharing, remixing, and monetising features—emerging not from the logic of finished works but from the modular ecosystem of processual contributions. The infrastructure already supports all compositional functions necessary for such applications to operate. What remains is the translation of this potential into practical environments—designed for artists, not only developers.

Moreover, new kinds of creative economies could emerge around the contribution and activation of compositional logic. In contrast to conventional art markets, which revolve around the trade of finalised objects, the DCN infrastructure enables the circulation of processes, tools, and operational building blocks. Financial conditions embedded in features may compensate upstream contributors each time their logic is reused. Curators could emerge not as exhibitors of works, but as orchestrators of feature ecosystems—assembling and maintaining libraries of generative logic with specific aesthetic orientations. In such an environment, the notion of the artist might itself blur, giving rise to hybrid roles: curator-programmers, composer-infrastructuralists, agent-facilitators.

In this light, PTs may extend far beyond compositional practice and serve as a foundational infrastructure for the management and execution of modular assets in future metaverses. Their ability to encode compositional logic, track provenance, define activation conditions, and structure recursion makes them ideally suited to model not only artistic processes but complex interactions in immersive digital environments. A PT could just as easily define the behaviour of a virtual object, a game mechanic, or a dynamic score as it does a musical phrase. As metaverses increasingly demand programmable and interoperable asset architectures, PTs offer a compelling solution—unifying symbolic, procedural, and relational logic in a standardised, extensible format.

Finally, the availability of a modular and programmable compositional infrastructure opens new possibilities for realising visionary projects that until now have remained speculative. One such project is *Hypermusic Experiment 0.9* by Einar Torfi Einarsson—a system that operates as an "infinite notation-machine" and generates decentralised,

nonlinear, multipiece scores from internal processes, performer interpretations, and external data inputs (Einarsson, 2024, 130–141). The project's emphasis on networked interactivity and API-driven responsiveness aligns well with the capabilities of the DCN model. By using DCNs, *Hypermusic Experiment 0.9* could now be instantiated as a living, compositional ecosystem—where each partial-score, external trigger, or post-performance response becomes a PT within a distributed graph of operations.

A second example comes from Kosmas Giannoutakis, who similarly envisions decentralised technologies as enablers of collaborative and sustainable music-making practices, rooted in mutualism and the commons, and structurally "rhizomatic" (107). Giannoutakis proposes a blockchain-based framework for collaborative electroacoustic music-making through live coding (2024, 1). His approach, which integrates decentralised ledger technologies to enable transparent authorship and distributed agency, "aims to facilitate collaborative music-making and also challenge traditional notions of authorship and the creative process" (Giannoutakis, 2024, 70). Here too, DCNs offer a ready-made environment: every code snippet could be encoded as a transformation, every live-coded gesture as a PT, and every collaboration as an evolving graph of modular relations across users and time.

What these projects have envisioned conceptually can now be encoded, executed, and shared via a compositional system that supports modularity, provenance, and recursion at every level. In this sense, DCNs do not only model a new approach to composing; they operationalise a new phase in artistic research—one in which speculative infrastructures can finally come to life.

From Composition to Ecosystem: Worlding through Performative Transactions

As the DCN accumulates new PTs, it becomes an evolving ecosystem of interrelated compositional logics. Every contribution extends the operational vocabulary of the network; every new PT deepens the graph of executable relations. What emerges is a compositional space that continually reorganises itself in response to new processes.

When a user contributes a new feature or transformation, they are are modifying the space of possibilities within which other users, agents, or systems can operate. A new transformation might offer a novel mode of variation; a new condition might introduce a procedural rule that filters which kinds of transactions are permitted. In

each case, the contribution has a world-making quality: it alters the structure of the compositional environment.

This shift—from composing outputs to composing systems—marks a deeper transformation. The network becomes a space in which artists, developers, and AI agents are not just producing artefacts, but co-constructing an evolving infrastructure. The DCN functions as a recursive system allowing compositional logic to restructure itself. Because each component is executable, reusable, and referential, new contributions expand the space of interaction, enabling increasingly complex operations and feedback loops across agents and levels. Over time, this operational space gives rise to a form of compositional worlding. Every PT intervenes in the compositional system as such—establishing new relations, exposing new variables, or defining new modes of interaction. These interventions accumulate, forming a topology of operational possibilities that shapes the way future works are conceived and executed. Users entering the system encounter a dynamic ecosystem of interdependent components—each representing a decision, a strategy, a logic that can be activated or reconfigured in their own practice.

While artists can use this system to describe and operationalise the intricate operations of whole artistic ecosystems, the network can also be used more pragmatically in everyday's compositional practice. Composers can "world" small ecosystems of compositional procedures as reusable vocabulary, which might probably be the most frequently used way of interacting with the system. They could for instance, create a space of reusable generative processes for a given compositional style, or for a given set of compositional procedures. Importantly, they can seemlesly discover and reuse such components created by other human and non-human actants. This worlding is embedded in the infrastructure. The network exposes all compositional processes as programmable interfaces, allowing users and agents to write themselves into the system by defining how things unfold. In doing so, they participate in generating the operational conditions under which content is produced. And because these conditions are stored, indexed, and executable, they persist beyond their initial context, enabling other agents to inhabit, modify, or recombine them.

In this model, creative practice becomes infrastructural: the act of composing is inseparable from the design of the system through which composition takes place. Each contribution is simultaneously an artistic gesture and a reconfiguration of the world in which future gestures will occur. This is the compositional significance of

Performative Transactions. They encode not only a process but a relation—a way of structuring how processes reference, transform, and activate one another. They define the logic of compositional inheritance and mutation, making it possible to trace how ideas evolve, diverge, or converge across time and across agents. And because they are executed within a programmable, decentralised framework, they allow these relations to persist as part of the operational reality of the network.

In such a system, composition becomes a matter of ecology: a practice of cultivating, navigating, and expanding a network of interdependent processes. The composer no longer authors isolated forms, but intervenes in an environment—planting seeds, grafting branches, pruning pathways. The artwork is not the final output, but the ongoing system through which outputs are generated, shared, and transformed. The result is a shift from composition as expression to composition as ecosystem-building: a recursive, collective, programmable space in which the infrastructure itself becomes the object of creative work.