System Workflow

Table of Contents

---

1. Overview

1.1 End-to-End Flow
1.2 Required Inputs

2. Creating a Lexicon (Scribe)

2.1 Input Sources
2.2 Configuration
2.3 Output

3. Inspecting a Lexicon (LAT)

3.1 Loading a Lexicon
3.2 View Modes
3.3 Weight Analysis

4. Preparing a Transformation (Engine)

4.1 Entering a Mnemonic
4.2 Parsing Behavior
4.3 Active Key Assembly

5. Executing a Transformation

5.1 Initial Index Selection
5.2 Forward Pass
5.3 Reverse Pass
5.4 Directional Behavior

6. Minimal Working Example

6.1 Example Lexicon
6.2 Example Mnemonic
6.3 Example Input Data
6.4 Step-by-Step Result

7. Variations

7.1 Repeated Words
7.2 Missing Words
7.3 Large Lexicons vs Small Lexicons

8. Failure Characteristics

8.1 No Error States
8.2 Always Produces Output
8.3 Compression Behavior
8.4 Implications for Validation

9. Workflow Patterns

9.1 Experimental Use

9.2 Reproducible Use

9.3 Airgapped Use

---

1. Overview

This document describes the complete operational workflow of the Vizini2 system, from lexicon generation through transformation execution.

The system is composed of three primary components:

- Scribe - Generates the VLex (Vizini Lexicon) from input sources
- LAT (Lexicon Analysis Tool) - Provides inspection and analysis of the VLex
- Vizini2 - Performs the transformation process using a mnemonic and runtime parameters

Each component operates independently, but all rely on a shared data model and consistent terminology.

The workflow is deterministic: given identical inputs, the system will always produce identical output.

1.1 End-to-End Flow

At a high level, the Vizini workflow proceeds as follows:

1. **Lexicon Creation (Scribe)**

Input data is processed to generate a VLex containing structured transformation data (Chunks composed of S-boxes).

2. **Lexicon Inspection (LAT)** (optional but recommended)

   The VLex is analyzed to understand structure, weight distribution, and potential mnemonic behavior.
3. **Mnemonic Definition (LAT Analyze) (Vizini2 Input)**

   A human-readable mnemonic is selected to define how data from the VLex will be assembled.
4. **Active Key Assembly (Vizini2)**

   The mnemonic is parsed and expanded into a linear Active Key composed of ordered S-boxes.
5. **Transformation Execution (Vizini2)**

   Input data is processed through a two-pass transformation (forward and reverse), producing output data.

1.2 Required Inputs

To perform a transformation, the system requires the following inputs:

- VLex (Vizini Lexicon)

  The exact SQLite database used to source transformation data.

  Any variation in the lexicon results in a completely different transformation.
- Mnemonic

  A sequence of words used to select and assemble Chunks from the VLex.

Order, repetition, and composition directly affect the Active Key.

- Initial Indices and Parameters

  Runtime values that determine traversal behavior within the Active Key.

  Even small variations produce entirely different outputs.
- Input Data

  The raw data to be transformed.

2. Creating a Lexicon (Scribe)

The Scribe component is responsible for generating a VLex (Vizini Lexicon) from input data.

This process transforms human-readable input into structured transformation data composed of Chunks and S-boxes. The resulting lexicon serves as the foundation for all subsequent operations.

The process of building a lexicon from a medium-sized .txt file may take several minutes to complete. Larger inputs, higher depth, and heavier chunks can increase processing time significantly, potentially exceeding an hour.

2.1 Input Sources

Scribe accepts two primary input methods:

- **Raw Text**

  Bulk input data (e.g., documents, word lists, or arbitrary text) is parsed into individual words. These words become the basis for Chunk generation.
- **Manual Entry**

Individual words or phrases can be entered directly. This allows precise control over lexicon composition and is useful for curated or minimal datasets.

2.2 Configuration

Scribe uses configurable parameters to control how input data is expanded into structured transformation data.

Depth (Iteration)

Defines how many distinct iterations are generated for each word.

- Each iteration produces a unique Chunk
- Higher depth increases variability when words repeat in a mnemonic
- Depth is typically defined using a dice-math expression
- The expression is reevaluated for each input word

Weight (S-Box Count)

Defines how many S-boxes are generated per Chunk.

- Higher weight increases transformation complexity and traversal length
- Weight directly contributes to total Active Key size
- Weight is also defined using a dice-math expression
- The expression is reevaluated independently for each Chunk

Dice Math (DMER)

Depth and Weight values are resolved using a Dice Math Expression Resolver (DMER).

This system evaluates expressions such as:

- `1d6`
- `2d8 + 3`

- `(1d4 + 1) * 2`
- `((2d3)d(4d5)*2)+(1d3)`
- `7`

Key properties:

- The `d` operator represents dice rolls (`XdY` = roll `X` dice with `Y` sides)
- Expressions support addition (`+`), subtraction (`-`), and multiplication (`*`)
- Parentheses allow nested evaluation
- Results are computed at generation time using a cryptographically secure random source

Each evaluation produces a concrete integer used for Depth or Weight during lexicon creation.

Duplicate Handling

Defines how repeated words are handled during input processing:

- **Append**

  Adds new Chunks alongside existing ones, increasing total iterations
- **Ignore**

  Skips duplicates, preserving the original entry
- **Replace**

  Overwrites existing entries with newly generated Chunks

This setting directly affects lexicon structure and mnemonic behavior.

2.3 Output

The result of the Scribe process is a fully constructed VLex containing structured transformation data and metadata.

Generated Chunks

- Words are mapped to one or more Chunks
- Each Chunk contains:
    - A defined number of S-boxes (Weight)
    - A specific iteration index (Depth)
- Chunks are the fundamental units used during Active Key assembly

Metadata

The VLex includes *optional* metadata fields for identification and documentation:

- Title

  A human-readable name for the lexicon
- Author

  The creator's name or pseudonym
- Author's Notes

  Freeform text for comments, context, or usage notes

Metadata does not affect transformation behavior and is stored purely for usability and reference.

3. Inspecting a Lexicon (LAT)

The Lexicon Analysis Tool (LAT) provides visibility into the internal structure of a VLex.

It allows users to inspect how words, iterations, and Chunk weights are distributed, enabling informed decisions when constructing mnemonics or evaluating lexicon quality.

LAT does not modify the VLex. It is a read-only tool used for inspection and analysis.

3.1 Loading a Lexicon

A VLex is loaded directly into LAT from its SQLite file.

Once loaded, LAT parses the lexicon structure and exposes:

- Available words
- Iteration counts (Depth)
- Chunk composition and associated weights

The integrity of the lexicon is assumed. LAT does not validate correctness beyond structural readability.

3.2 View Modes

LAT provides multiple ways to view lexicon data, depending on the level of detail required.

Aggregate Mode (`*`)

Displays all iterations of a word as a single grouped entry as `word.depth`.

- Shows total weight across all iterations
- Useful for understanding overall contribution of a word
- Helps identify dominant or weak entries in the lexicon

This mode is best suited for high-level analysis and mnemonic planning.

Granular Mode (`word.iteration`)

Displays individual iterations of a word as `word.iteration`.

- Each entry corresponds to a specific Chunk
- Shows exact weight per iteration
- Allows inspection of variability between iterations

This mode is useful when precise control or detailed analysis is required.

3.3 Weight Analysis

LAT provides tools for understanding how weight is distributed across the lexicon and how it may affect transformation behavior.

Per-Word Weight

- Displays the total weight associated with a word across all iterations
- Helps identify disproportionately heavy or light entries
- Useful for balancing lexicons or refining input sources

Mnemonic Weight Estimation

- Estimates the total Active Key weight for a given mnemonic
- Based on selected words and their available iterations
- Provides a preview of transformation scale and traversal length

This allows users to:

- Anticipate performance characteristics
- Avoid excessively large Active Keys
- Tune mnemonics for desired complexity

4. Preparing a Transformation (Vizini2 Engine)

The Vizini2 Engine prepares a transformation by converting a human-readable mnemonic into a structured Active Key.

This process is deterministic and depends entirely on the provided VLex, mnemonic, and parameters.

4.1 Entering a Mnemonic

The mnemonic is a sequence of words entered by the user.

- Words are interpreted in order
- Repetition is allowed and meaningful
- Each word must exist in the VLex to contribute to the Active Key
- Words absent from the VLex are ignored during parsing and may be used as filler for readability, memorability, and natural language phrasing

The mnemonic defines both:

- Selection - which words are used
- Structure - the order in which their associated Chunks are assembled

Even small changes (word choice, order, or repetition) result in completely different Active Keys.

4.2 Parsing Behavior

The Engine parses the mnemonic sequentially from beginning to end.

For each word:

- The corresponding entry is located in the VLex
- The Chunk associated with the word's iteration of occurrence is selected (wrapping behavior applies if occurrences exceed available iterations)
- Selected Chunks are passed to the assembly stage

Key properties:

- Parsing is deterministic given identical inputs
- Word order is preserved exactly
- Repeated words are processed independently
- Missing words do not produce errors, but do not contribute meaningful structure

No normalization, correction, or validation is performed during parsing.

4.3 Active Key Assembly

The Active Key is constructed by appending all selected Chunks in sequence.

- Each Chunk contributes its full set of S-boxes without modification
- Chunks are appended in the exact order determined by the mnemonic
- No reordering, merging, or optimization is performed

The result is a linear sequence of S-boxes, which is traversed via an index during transformation.

Resulting Properties:

- The Active Key is fully deterministic given identical inputs
- Total length is determined by the combined weight of all selected Chunks
- Structural patterns in the mnemonic directly translate into transformation structure

All Chunk selection and ordering decisions are completed before transformation begins. The Active Key remains fixed for the duration of the process.

The mnemonic does not encode data directly; it defines how structured data is assembled and applied.

5. Executing a Transformation

Once the Active Key has been constructed, the Vizini2 Engine performs the transformation process.

This process operates on input data using a dynamic traversal of the Active Key. Behavior is influenced by both the current index position and the evolving output.

The transformation consists of two passes:

- A forward pass (beginning → end)
- A reverse pass (end → beginning)

The order of these passes and the lookup behavior depend on the transformation direction.

5.1 Initial Index Selection

Two independent indices are required:

- Forward Pass Index
- Reverse Pass Index

Each index:

- Determines the starting position within the Active Key for its respective pass
- Must be within the range `0` to Total Weight - 1
- Is normalized using modulo arithmetic if necessary

These values are user-provided inputs.

Even small changes to either index result in completely different transformation behavior.

Both indices must be reproduced exactly to achieve a reversible transformation. There is no mechanism to infer or recover index values from output data.

5.2 Forward Pass

The forward pass processes data from beginning to end.

For each unit of data:

1. The current index selects an S-box from the Active Key
2. The value is transformed using that S-box
3. The result is written as the output value for that step
4. The index is updated[^1] using the value interpreted as ciphertext for the current operation
    - During forward transformation, this corresponds to the output value
    - During reverse transformation, this corresponds to the input value (treated as ciphertext in the inverse context)

Key properties:

- Index updates are ciphertext-dependent
- Each step influences subsequent traversal
- The entire data stream becomes statefully linked

5.3 Reverse Pass

The reverse pass processes data from end to beginning.

For each unit of data:

1. The current index selects an S-box from the Active Key
2. The value is transformed using that S-box
3. The result is written as the output value for that step
4. The index is updated[^1] based on the value interpreted as ciphertext for the current operation

    - During forward transformation, this corresponds to the output value
    - During reverse transformation, this corresponds to the input value (treated as ciphertext in the inverse context)

Key properties:

- Reinforces diffusion across the dataset
- Introduces bidirectional dependency
- Ensures early and late data influence each other

5.4 Directional Behavior

The Vizini2 Engine supports bidirectional transformation using the same underlying process.

Forward Transformation

- Pass order:
    1. Forward pass
    2. Reverse pass
- Uses standard S-box lookups

Reverse Transformation

- Pass order:
    1. Reverse pass
    2. Forward pass
- Uses inverse S-box lookups

Independent index initialization ensures that both forward and reverse transformations have well-defined starting states, regardless of execution order.

Transformations are performed using two buffers. Each pass reads from a source buffer and writes results to a separate working buffer. Between passes, the buffers swap roles so that the next pass operates on the fully transformed output of the previous pass.

Resulting Behavior

- Each pass operates with its own independently initialized index
- Index state does not carry between passes
- The transformation is stateful within each pass
- The two-pass structure creates global interdependence across the dataset

---

[^1]: The index is updated as follows:

index := (index + value) mod Total Weight

Where:

- value is the quantity interpreted as ciphertext for the current operation
- Total Weight is the length of the Active Key (total number of S-boxes)

---

6. Minimal Working Example

For brevity and readability, the following examples are stripped down to only what is necessary to explain the process.

For these examples, we will use a lexicon with the filename `Test.VLex`

6.1 Example Lexicon

Inside `Test.VLex` are 2 tables

Metadata

Title	Test Sample
Author	Test Monkey
Notes	This is a Test of the Vizini2 Cipher Engine This is only a test

Chunks

Word	Iteration	Weight
the	1	5
the	2	4
cat	1	7
Bird	1	2
...	...	...

From `the.1` in the `Chunks` table, the following S-box is derived:

Input	Output
0x00	0x63
0x01	0x42
0x02	0x7C
...	...
0xFF	0x16

6.2 Example Mnemonic

When using `Test.VLex` with the mnemonic `The cat in the window sees the bird`

The words `in` and `window` are absent and ignored.

The word `the` appears 3 times in the mnemonic, but has a depth of only 2, so the 3rd instance of `the` uses the first iteration from the lexicon due to modulo math.

The mnemonic resolves as: `the.1 cat.1 the.2 sees.1 the.1 bird.1` with weights of 5 + 7 + 3 + 13 + 5 + 2, for a total weight of 35.

Initial Indices:

- Forward Pass Index: 2
- Reverse Pass Index: 7

6.3 Example Input Data

For demonstration purposes, the input data will be a text file containing `Kittens are cute.`

```
0x4B 0x69 0x74 0x74 0x65 0x6E 0x73 0x20 0x61 0x72 0x65 0x20 0x63 0x75 0x74 0x65 0x2E
```

6.4 Step-by-Step Result

At each step, the S-box is selected using: `S-box := ActiveKey[index mod Total Weight]`

Using the `Forward (Encrypt)` direction will result in the following behavior.

Forward Pass:

Step	Index	Input	Output	Index + Ciphertext	Updated Index
0	2	0x4B	0xBE	192	17
1	17	0x69	0x03	20	20
2	20	0x74	0xC2	214	4
3	4	0x74	0xF3	247	2
4	2	0x65	0x70	114	9
5	9	0x6E	0xDD	230	20
6	20	0x73	0x97	171	31
7	31	0x20	0xAD	204	29

8	29	0x61	0xEA	263	18
9	18	0x72	0xDF	251	6
10	6	0x65	0xED	243	33
11	33	0x20	0x17	56	21
12	21	0x63	0x8B	160	20
13	20	0x75	0x9A	174	34
14	34	0x74	0x52	116	11
15	11	0x65	0x13	30	30
16	30	0x2E	0x20	↠	N/A

↠ indicates the final step, after which no further index update is required.

Reverse Pass (operating on output of Forward Pass):

Step	Index	Input	Output	Index + Ciphertext	Updated Index
0	7	0x20	0xBD	196	21
1	21	0x13	0xE9	254	9
2	9	0x52	0x07	16	16
3	16	0x9A	0xB6	198	23
4	23	0x8B	0xCD	228	18
5	18	0x17	0xE8	250	5
6	5	0xED	0x04	9	9
7	9	0xDF	0xC0	201	26
8	26	0xEA	0x90	170	30
9	30	0xAD	0xC3	225	15
10	15	0x97	0x83	146	6
11	6	0xDD	0xB1	183	8
12	8	0x70	0xEF	247	2
13	2	0xF3	0x51	53	18
14	18	0xC2	0x11	35	0
15	0	0x03	0xDC	220	10
16	10	0xBE	0xD1	↠	N/A

Final Output (Hex):

`0xD1` `0xDC` `0x11` `0x51` `0xEF` `0xB1` `0x83` `0xC3` `0x90` `0xC0` `0x04` `0xE8` `0xCD` `0xB6` `0x07` `0xE9` `0xBD`

Repeating this process in Reverse (Decrypt) mode with the same mnemonic and initial indices will reproduce the original input data exactly.

7. Variations

The Vizini2 Engine permits flexible mnemonic construction. The following variations describe how different mnemonic characteristics influence transformation behavior.

7.1 Repeated Words

Words may be repeated within a mnemonic. Each occurrence is processed independently and contributes to the Active Key.

- Each repetition advances the iteration index for that word
- When the number of occurrences exceeds available iterations, selection wraps using modulo arithmetic
- When repetitions exceed the word's depth, Chunk selection wraps, causing previously used Chunks to repeat

Implications:

- Repetition increases the total weight of the Active Key
- Repetition may introduce structural patterns into the transformation, especially with shallow depth
- Even identical words may contribute different Chunks depending on depth and occurrence

7.2 Missing Words

Words that are not present in the VLex are ignored during parsing.

- No Chunk is selected for missing words
- Parsing continues without interruption
- Word order of valid entries is preserved

Implications:

- Mnemonics may include filler words for readability or memorability
- Missing words do not affect the transformation beyond their absence
- Removing or adding unknown words does not change the Active Key

7.3 Lexicon Scale and Structure

The behavior of the Vizini2 Engine is influenced by multiple independent properties of the VLex. “Large” and “small” lexicons may differ along several dimensions, each with distinct effects.

Vocabulary Size (Number of Words)

The total number of unique words available in the VLex.

Effects:

- Larger vocabularies allow more expressive and varied mnemonics
- Reduces likelihood of repeated word usage
- Increases flexibility without requiring structural reuse

Implications:

- Larger vocabularies support more natural language phrasing
- Smaller vocabularies encourage reuse and repetition of words

Iteration Depth (Per-Word Variants)

The number of iterations (Chunks) available for a given word.

Effects:

- Greater depth allows repeated use of a word to resolve to different Chunks more often
- Reduces frequency of modulo wraparound
- Increases variation when the same word is reused

Implications:

- Shallow depth leads to rapid reuse of the same Chunk
- Deep words provide more structural diversity under repetition

Chunk Weight (S-box Count per Chunk)

The number of S-boxes contributed by each Chunk.

Effects:

- Higher weights increase the total size of the Active Key
- Larger Active Keys increase the effective traversal space
- Influences how far the index moves before wrapping

Implications:

- Heavier chunks produce longer traversal cycles
- Heavier chunks can produce a large Active Key with a shorter mnemonic
- Lighter chunks result in tighter, more frequently repeating traversal patterns

Combined Effects

These properties interact to determine the overall structure of the Active Key.

- A lexicon with many words but shallow depth favors breadth over variation under repetition
- A lexicon with fewer words but high depth favors reuse with variation
- Increasing chunk weight expands the traversal space regardless of vocabulary size

Result:

The structure and behavior of the transformation depend not only on the mnemonic, but on how the lexicon distributes vocabulary, depth, and weight.

There is no single definition of an "optimal" lexicon; effectiveness depends on how these properties are balanced for the intended use case.

8. Failure Characteristics

The Vizini2 Engine is designed such that all valid inputs produce output. As a result, traditional error states are largely absent. This section describes behaviors that may be mistaken for failure but are inherent to the system's design.

8.1 No Error States

The engine does not define internal failure conditions during transformation.

- All inputs are processed as provided
- Invalid or unrecognized mnemonic elements are ignored rather than rejected
- No structural validation is performed on the resulting Active Key

Implications:

- The system will not signal incorrect or unintended configurations
- Misconfigured inputs still produce valid output
- Responsibility for correctness lies with the user

8.2 Always Produces Output

For any given combination of input data, mnemonic, and lexicon, the engine will produce output data of equal length.

- No input is rejected during execution
- Output is always generated when the Active Key contains at least one Chunk

Implications:

- Output existence does not imply correctness
- Incorrect configurations may silently produce unintended results
- There is no built-in mechanism to distinguish valid transformations from invalid ones
- If the mnemonic produces an empty Active Key, execution is not performed and the user is prompted to modify the mnemonic

8.3 Compression Behavior

Encrypted output is not guaranteed to be incompressible.

While the system aims to produce high-entropy transformations, certain combinations of:

- Lexicon
- Mnemonic
- Starting indices
- Input data

may introduce incidental structure in the output.

As a result:

- In rare cases, encrypted data may be further compressible

- This behavior is not consistent or predictable
- Compression effectiveness should not be used as an indicator of correctness or security

This phenomenon reflects the system's stateful and structured transformation model rather than a failure condition.

8.4 Implications for Validation

The engine is deterministic, but not self-validating.

Because the system lacks explicit error states, validation must be performed externally.

- Successful execution does not confirm correctness
- Output cannot be assumed valid without independent verification
- Identical inputs and configuration are required to reproduce or undo results

Recommended approaches:

- Use known test vectors to confirm expected behavior
- Verify transformations through round-trip testing (transform → inverse transform)
- Track and validate mnemonic and lexicon integrity outside the engine

9. Workflow Patterns

The Vizini2 Engine supports multiple usage patterns depending on user goals. These patterns describe common approaches to balancing flexibility, reproducibility, and operational constraints.

9.1 Experimental Use

Experimental workflows prioritize exploration and variation over reproducibility.

- Mnemonics may be created or modified freely
- Lexicons may change between runs
- Starting indices may be arbitrary or omitted

Characteristics:

- High variability in output
- Low reproducibility
- Useful for discovery, testing, and analysis

Implications:

- Results are difficult to reproduce without full configuration tracking
- Small changes can produce significantly different outputs
- Not suitable for scenarios requiring consistency

9.2 Reproducible Use

Reproducible workflows prioritize consistency and determinism.

- Mnemonics are fixed and recorded exactly
- Lexicon content and structure are preserved
- Starting indices are explicitly defined

Characteristics:

- Deterministic output for identical inputs
- High reproducibility
- Stable transformation behavior

Implications:

- Any deviation in mnemonic, lexicon, or indices alters the result
- Configuration must be preserved alongside the data
- Suitable for verification, storage, and repeatable processes

9.3 Airgapped Use

Airgapped workflows prioritize isolation and control over external dependencies.

- Lexicons are stored and maintained locally
- Mnemonics are managed without external systems
- Execution occurs in an isolated environment

Characteristics:

- No reliance on networked resources
- Reduced exposure to external observation or interference
- Fully self-contained operation

Implications:

- All required data must be transferred manually
- Version control of lexicons and configurations becomes critical
- Loss of configuration data may prevent reproduction of results

These workflow patterns are not mutually exclusive. Effective use of the Vizini2 Engine often involves combining elements of each approach based on operational needs.