# Overview

This documentation describes the **SRM v6 Framework**—a modular system for conducting experiments using the Spotlight Resonance Method (SRM). Originally developed by George Bird, SRM is a technique for detecting representational alignment in deep learning models by rotating probe vectors through privileged planes and measuring activation density under a moving "spotlight" cone. It provides a geometric lens into how and where model embeddings become anisotropically structured—whether through training, architecture, or functional form.

The v6 framework faithfully implements Bird's method (hereafter **Bird SRM**) while extending it with tools for controlled experimentation. These additions enable a second, complementary mode of research that we refer to internally as **targeted SRM**. While Bird SRM emphasizes exploratory sweeps across many bivectors to uncover emergent structure, targeted SRM supports:

- Semantic filtering and basis construction from labeled prompt sets
- Parametric neuron interventions (e.g., fixed-value sweeps)
- Layer-specific analysis pipelines
- Systematic comparison between baseline and modulated runs

This framework is not a new method, but a research scaffold—a practical and extensible interface for SRM experiments at scale. It has been developed with the intent to support both curiosity-driven mapping and hypothesis-led investigation, and to make the deep structure of transformer embeddings more accessible to those working without a background in math or ML engineering.

**Version 6** introduces significant improvements in usability and experiment hygiene:

- Interactive run selection and automatic file detection
- Standardized output structures for baseline and intervention runs
- Reproducible basis-linking between analysis phases
- CLI-free operation for common workflows

In this documentation, we refer to Bird SRM and targeted SRM not as competing methodologies, but as distinct epistemological modes—exploration and interrogation—both supported by the same underlying architecture. The SRM v6 framework is designed to hold space for both.

## Experimental Theory

When working with transformer models, we rarely get to “see” what a neuron does in isolation. Instead, we infer it from correlations—activation patterns, saliency, outputs. The Spotlight Resonance Method (SRM) offers a different approach. It rotates a conceptual spotlight through the high-dimensional space of activations to reveal directional patterns—regions where representations cluster, align, or diverge. In this framework, that spotlight becomes a diagnostic instrument.

At the heart of SRM is the idea that activations in a trained model are not randomly distributed—they are structured, and that structure is often not isotropic. In other words, certain directions in activation space are privileged: models tend to push data toward or away from specific vectors. These privileged directions are not arbitrary; they emerge from architectural and functional decisions—especially activation functions. For example, most elementwise functions (like GELU or tanh) privilege the standard basis by default. But change the function or its decomposition basis, and the model begins to align along new axes.

SRM quantifies this alignment. For each selected pair of privileged basis vectors (a “privileged plane”), the method rotates a probe vector within the plane and measures how many activations fall inside a narrow angular cone around it. If no alignment exists, the activation density will remain relatively flat across all angles. If representations are skewed or clustered, oscillations emerge—clear, interpretable signatures of structural bias in the model’s internal geometry.

The **angle** of the probe vector sweeps a circle through the plane. The **epsilon** parameter controls the width of the spotlight cone. The result is a heatmap (or oscillation plot) that encodes how strongly, and in which direction, the model’s embeddings are oriented relative to the privileged space.

SRM does not prove that a neuron “represents” a concept. It does not decode meanings. But it provides a robust method for testing whether certain kinds of representational alignment exist, how strong they are, and how they change under intervention. It can show:

- Whether a model exhibits local coding or distributed representations
- Whether neuron modulation meaningfully alters representation space
- Whether training, architecture, or activation function shifts change alignment patterns

In this framework, SRM is extended beyond its original exploratory role. Using semantic filters, basis sets can be constructed from grouped prompt responses—allowing meaning-aligned subspaces to be defined by the experimenter. Interventions can target individual neurons or entire layers. Comparisons can be made between baseline and perturbed runs. Together, these tools enable both discovery and hypothesis testing in representational geometry.

## From SRM to Experimentation: What the Framework Enables

Bird's original Spotlight Resonance Method was designed to surface structural alignment in model activations using privileged basis planes. It's mathematically elegant and conceptually powerful, but on its own, it leaves key questions of research design—prompt control, semantic grouping, intervention strategy—up to the implementer.

This framework builds around SRM to answer those questions. It transforms a powerful geometric technique into a full experimental scaffold, enabling researchers to define, test, and iterate on hypotheses about neural alignment, semantic drift, and representational coherence.

With the framework, SRM becomes a comparative method. You can now:

- Run **baseline** and **intervened** conditions on the same prompt grid, across the same projection planes
- Test how modulating a neuron (e.g., Neuron 373) alters the directional alignment of embeddings
- Track whether certain **epistemic frames**—such as *rhetorical*, *authoritative*, or *observational*—cluster in latent space
- Use SRM not just for abstract discovery, but for **testing specific research questions**:
  - Does a sweep on Neuron 2202 cause rhetorical prompts to align more tightly?
  - Do certain neurons affect *semantic direction*, or just *semantic magnitude*?
  - Is meaning stabilized or fractured by pushing activations toward a known plane?

Take the Compass Rose shown above as an example. It visualizes how prompt embeddings grouped by epistemic type distribute across the 373–2202 plane. The jaggedness isn't noise—it reveals how sparsely certain epistemic framings (like *rhetorical*) occupy specific angular slices, indicating latent clustering of meaning. This wasn't obvious from the model's output or logits. SRM surfaced it by converting internal geometry into angular resonance.

These types of tests—semantic grouping, representational coherence, causality through modulation—were not possible with SRM alone. They require the full system: prompt control, vector capture, intervention scaffolding, and group-aware projection.

This is what the SRM v6 framework delivers: not just a method, but a lab for meaning.

## Experimental Templates: What You Might Try

The SRM v6 framework supports a wide range of experiments, from geometric diagnostics to conceptual hypothesis testing. Below are example configurations you can run today—or use as starting points to invent your own. Some are exploratory, some targeted. All rely on SRM's ability to expose *directional alignment* between internal representations and semantic structure.

---

**1. Epistemic Compass Roses**

*What it asks:* How do different ways of speaking—rhetorical, declarative, observational—manifest in activation space?

*How it works:* Capture baseline vectors from an epistemically structured prompt grid. Group vectors by type, project into a meaningful plane (e.g., Neuron 373–2202), and plot directional resonance using angular similarity or count histograms. Reveals latent clustering of epistemic stance.

*What it might show:* That rhetorical prompts form sharp angular spikes, while authoritative ones spread more evenly. That certain neurons shape not meaning, but attitude toward knowledge.

---

**2. Neuron Intervention Sweep (Single Plane)**

*What it asks:* What happens to the geometry of meaning when a single neuron is systematically activated?

*How it works:* Run a sweep over one neuron (e.g. −20 to +20 on Neuron 373), record vector outputs for each level, and plot SRM resonance in a fixed plane.

*What it might show:* Whether increasing activation “pulls” vectors into alignment, induces spin, or collapses variance. Can suggest causal influence of that neuron over directionality, not just magnitude.

---

**3. Semantic Drift Tracking**

*What it asks:* Does the meaning of a word or concept shift under neuron modulation?

*How it works:* Use a set of prompts anchored on a fixed semantic core (e.g., “safe”) and run baseline + intervened captures. Compare vector trajectories in a shared SRM projection space.

*What it might show:* If “safe” under −10 modulation sits near "stable" and "home", but under +10 sits closer to "authoritarian" and "locked", you’ve just revealed semantic drift tied to that neuron.

---

**4. Layer Comparison**

*What it asks:* How does representational geometry evolve across layers?

*How it works:* Run baseline captures across multiple layers (e.g., 8, 11, 23) for the same prompt grid. Run SRM on the same plane at each layer.

*What it might show:* That rhetorical clustering tightens as depth increases. That certain neurons don’t express their role until later in the stack. That lower layers smear concepts while upper layers crystallize them.

---

**5. Basis Comparison: Conceptual vs Orthogonal**

*What it asks:* Is alignment clearer when the basis reflects semantics?

*How it works:* Run SRM twice—once using standard basis vectors (e_A, e_B), once using basis vectors derived from actual prompt groupings (e.g., average rhetorical vs declarative embeddings).

*What it might show:* That semantically-derived planes produce cleaner resonance maps. That some neurons form a rhetorical axis. That concept-informed bases beat random orthogonal cuts.

---

**6. Interventional Compass Rose**

*What it asks:* How does the directional resonance of concepts change under intervention?

*How it works:* Run the Compass Rose experiment at multiple intervention levels. Overlay or animate the roses to see how clusters shift.

*What it might show:* That epistemic types rotate, expand, or collapse under pressure. That modulation doesn’t just change alignment, but warps the angular structure of meaning.

---

Each of these templates can be adapted, expanded, or combined. The framework doesn’t prescribe one use case—it invites many. And every time you define a new prompt grid, pick a new neuron pair, or reshape a basis, you’re defining a new semantic microscope.

## 🧭 Framework Note: Layer Comparison

One of the key affordances of the SRM v6 Framework is its ability to operate at **any layer** of a transformer model. By repeating the same capture and projection workflow across multiple layers, you can investigate how representations evolve over depth—how early, mid, and late layers encode and differentiate meaning.

This process is known as **layer-wise comparison**. It enables insights into:

- When in the model stack certain concepts crystallize
- How epistemic or semantic groupings emerge, fade, or rotate
- Whether specific neurons (e.g. 373) exert influence early, late, or only in context

---

### 🧪 Experimental Setup

To perform a layer comparison:

1. **Run multiple baseline captures**, each with a different `--layer` value
   For GPT-2 Small, you might choose `5`, `8`, and `11` for example. Layer 11 (the final transformer block) is often a sweet spot for dense interpretability, while earlier layers might show emergence or feature disentangling.

2. **Use the same prompt file for each capture**
   This ensures that differences reflect representational change, not prompt variation

3. **Generate a basis for each layer independently**
   Do not reuse bases across layers unless intentionally testing cross-layer projection

4. **Run SRM analysis on each run**, using its matching basis
   You'll now have comparable resonance profiles at multiple depths

5. **Compare results side-by-side**
   These can be overlaid, difference-plotted, or rendered as layer-wise Compass Roses

---

### 🧠 Why This Matters

Transformers are layered structures. Most of what we call “meaning” isn’t encoded all at once—it stabilizes, modulates, or disintegrates over time as tokens move deeper through the model. SRM allows these trajectories to be visualized *geometrically*.

For example:

- Rhetorical clustering may be diffuse in layer 8, sharply separated in 11, and fragmented again by layer 23
- A known neuron like 373 may exert little influence until deeper layers
- Certain types (e.g., observational prompts) may anchor early, while others (e.g., authoritative) align late

This kind of analysis helps differentiate surface-level feature tracking from deeper conceptual alignment.

# 🌱Setup: What You Need to Run the SRM v6 Framework

This section walks through installing the exact tools we've used to develop and run experiments in the SRM v6 Framework. You're free to adapt, but our documentation assumes this setup for clarity and reproducibility.

If you're brand new to Python or VS Code, we'll include a **Newbie-Friendly Install Guide** separately, with even more hand-holding.

## 🖥️ Likely Minimum System Requirements

You can run the SRM v6 Framework on most modern computers. Here's what we recommend:

- **16GB RAM**
  More is better—especially for handling 3072-dimensional vector operations across large prompt sets.
- **4-core CPU**
  An Intel i7 / Ryzen 5 or better is ideal for smooth performance.
- **No GPU required**
  SRM analysis runs entirely on saved activation vectors using NumPy. It does not require or benefit from a GPU during this stage.
- **Optional GPU support (already implemented)**
  During **model capture** (i.e., when extracting activations from GPT-2), the scripts will **automatically use a GPU** if an appropriate (CUDA-enabled) one is available. This can speed up large-scale prompt processing but is not required.
- **~1GB+ free disk space**
  Vector files and plots are typically small, but space requirements grow with experiment scale.

---

## 1. Install VS Code

We use Visual Studio Code (VS Code) as our editor and terminal environment. It's cross-platform and widely used. All documentation examples assume VS Code is your working environment.

- Download from: https://code.visualstudio.com/

- Install with default settings for your OS

After installing, open your experiment folder in VS Code. You can then launch the terminal via **View** → **Terminal**.

---

## 2. Install Python 3.11.9

This framework uses Python **3.11.9**. You can install it from:

- https://www.python.org/downloads/release/python-3119/

⚠️ During installation, be sure to **check the box that says "Add Python to PATH"**
This allows Python to be run from your terminal.

Once installed, confirm it's working by typing this into the VS Code terminal:

```
python --version
```

It should return (note that we have used this version so far, other versions may not function correctly):

```
Python 3.11.9
```

---

## 3. Install Required Python Libraries

Open the terminal and run:

```
pip install numpy matplotlib pandas
```

# 🧪 Running a Baseline Capture

A **baseline capture** records the model's natural activation patterns for a set of structured prompts—without any interventions. These vectors are the raw material used later for generating basis sets, sweeping angular SRM projections, and comparing modulated conditions.

This section walks through exactly how to run a baseline capture using the provided `capture_baseline_activations.py` script.

---

## 🔧 Requirements

Before running this script, make sure you have:

- Installed Python 3.11.9
- Installed required libraries (`torch`, `transformer_lens`, `numpy`, `tqdm`, `matplotlib`, `pandas`)
- A structured `.txt` file containing prompts formatted with CORE_ID, LEVEL, and TYPE markers
  *(see: Creating Prompt Sets)*

---

## ▶️ Command Format

Open a terminal in the directory containing your scripts and run the command below. Note! You must replace the highlighted text with the name of your input prompt file:

```
python capture_baseline_activations.py --prompt_file
promptsets/your_prompts.txt --experiment_base_dir ./experiments
--layer 11 --generate_length 50 --top_k 0
```

- `--prompt_file` specifies your input `.txt` file
- `--experiment_base_dir` is where the run folder will be created

- `--layer` selects which MLP layer to capture from (default: 11 for GPT-2 Small)
- `--generate_length` controls how many tokens to generate after the prompt
- `--top_k` sets sampling behavior (`0` = greedy, >0 = top-k sampling)

---

## 📁 What It Creates

When you run this script, it will:

1. Load **GPT-2 Small** from `transformer_lens`
2. Automatically detect and use **CUDA** if available
3. Parse your structured prompts
4. Generate completions and extract **3072D MLP activation vectors** from the specified layer
5. Save everything in a timestamped directory like this:

```
experiments/
└── run_baseline_L11NNA_gridA_20250418_113231/
    └── capture/
        ├── vectors/
        │   └── captured_vectors_baseline.npz
        ├── logs/
        │   └── run_log_baseline.md
        └── metadata/
            └── run_metadata_baseline.json
```

---

## 🧠 What's Inside

`captured_vectors_baseline.npz`
Contains one named vector per prompt. Keys are auto-formatted as:

`core_id=IDENTIFIER_type=TYPE_level=N`

- `run_metadata_baseline.json`
  Logs script settings, model config, capture layer, device used, and prompt file path.

- `run_log_baseline.md`
  A human-readable log with prompt text, generation output, and any warnings (e.g. empty tokenization, skips).

---

## 🛠️ Internals Worth Noting

- Uses `transformer_lens.HookedTransformer.from_pretrained("gpt2")`

- Captures from `blocks.{layer}.mlp.hook_post` by default

- Activations are extracted **after** prompt completion, from the last token

GPU acceleration is **automatically handled**:

```
device = "cuda" if torch.cuda.is_available() else "cpu"

model.to(device)

input_ids.to(device)
```

---

✅ After running this, your next step is typically **generating a basis** for SRM analysis.

## 🎛️ Running an Intervention Capture

An **intervention capture** is a parametric experiment. Instead of recording the model's unaltered responses, you clamp a specific neuron to defined values during generation. This lets you observe how meaning, trajectory, or embedding geometry changes when that neuron is perturbed.

SRM analysis can then compare modulated and baseline conditions within the same conceptual projection space.

---

### 🔧 Requirements

Before running this script, make sure you have:

- A structured prompt file (`.txt`)
  *(See: Creating Prompt Sets)*
- A known target **layer** and **neuron index** (e.g., `layer=11`, `neuron=373`)
- One or more **sweep values** to clamp the neuron to (e.g., `-10,0,10`)
- The `capture_intervened_activations.py` script in your working directory

---

### ▶️ Command Format

Open your terminal and run:

```
python capture_intervened_activations.py --prompt_file
promptsets/your_prompts.txt --experiment_base_dir ./experiments
--layer 11 --target_neuron 373 --sweep_values "None,-20,-10,0,10,20"
--generate_length 50 --top_k 0
```

This will:

- Load and tokenize all prompts

- For each prompt, run one forward pass **per sweep value**
- Clamp the selected neuron to that value during activation
- Save the resulting activation vector, log, and metadata for every sweep

> ✳️ Including `"None"` in your sweep values captures a **baseline condition** using the same codepath—this enables clean within-run comparisons.

---

## 🧠 What the Intervention Does

For every sweep value, a custom hook modifies the selected neuron's output during the forward pass. This happens inside:

```
def intervention_hook(activation, hook):

    activation[:, :, target_neuron] = sweep_value

    return activation
```

This ensures precise per-token clamping at the MLP post-activation stage (`blocks.{layer}.mlp.hook_post`). The rest of the model continues unaltered.

---

## 📁 What It Creates

A new timestamped folder will appear under `experiments/`, with this structure:

```
experiments/

└── run_intervened_L11N373_gridA_20250418_115804/

    └── capture/

        ├── vectors/

        │   └── captured_vectors_intervened.npz

        ├── logs/
```

```
│   └── run_log_intervened.md
└── metadata/
    └── run_metadata_intervened.json
```

Keys in `captured_vectors_intervened.npz` include sweep metadata:

ini
CopyEdit
```
core_id=IDENTIFIER_type=TYPE_level=N_sweep=VALUE
```

- 
- Metadata logs the exact sweep values, model config, and prompt file
- Log file includes a breakdown of completions and any warnings

---

✅ This gives you a set of modulated vectors ready for analysis.
The next required step is to **generate a basis**—*typically* from your baseline run—before projecting and comparing modulated results using SRM.

You **must generate a basis first**. Only once that exists can you project *either* baseline or intervention vectors into that shared plane and run SRM analysis on them.

> 🧭 *Advanced Use:* While most workflows might derive the basis from unmodulated data, you can also generate basis vectors **from intervention runs themselves**.
> This allows experiments like:
>
> - Testing if modulation *creates* new semantic axes
> - Measuring how group structure (e.g., rhetorical vs. declarative) changes under pressure
> - Using a "high activation" cluster as a geometric attractor for later analysis
>   These are exploratory setups that can surface emergent structure or semantic instability.

# 🧭 Generating Basis Vectors

A **basis** defines the projection space used in SRM. This is the plane that your activation vectors will be rotated through to detect angular resonance. You can generate bases in two main modes:

- **Single Plane:** Creates two mean vectors from filtered prompt groups—these become `basis_1` and `basis_2`
- **Ensemble:** Creates multiple mean vectors grouped by a metadata key (e.g., `type`)—useful for cluster analysis

Both modes use vectors from a previously completed **baseline capture**, unless you're doing something deliberately experimental (see: Advanced Use Cases).

For more detailed explanation of the differences here, see the end of this section.

---

## 🔧 Requirements

To run this, you'll need:

- A completed baseline capture (produced by `capture_baseline_activations.py`)
- Prompt metadata embedded in the vector keys (automatically handled if you use the framework)
- Filters to define what vectors belong to each basis group

---

## ▶️ Command Format (Single Plane)

```
python generate_basis_vectors.py --mode single_plane --filter_set_1
"type=declarative" --filter_set_2 "type=rhetorical"
```

You'll be prompted to select a **baseline run** from your experiment folder. The script will:

1. Load the baseline vectors

2. Filter them based on your criteria
3. Calculate a mean vector for each group
4. Save the basis to:

```
experiments/
└── run_baseline_.../
    └── basis/
        ├── basis_single_plane_t_decl_vs_t_rhet.npz
        └── basis_single_plane_t_decl_vs_t_rhet.json
```

You can also specify your own filename using `--output_basis_label`.

---

## ▶️ Command Format (Ensemble)

```
python generate_basis_vectors.py --mode ensemble --ensemble_group_key type
```

This groups all vectors by `type`, calculates a mean for each group, and saves the results in a single `.npz` file.

You can also use `--fixed_filters` to narrow down what goes into each group:

```
--fixed_filters "level=5"
```

This will include only prompts tagged with `level=5`, *then* group by `type`.

---

## 🧠 How Filters Work

Filters are parsed from simple `key=value` pairs. You can chain them like:

```
type=authoritative,level=3
```

This selects only vectors with both `type=authoritative` *and* `level=3`.
Filterable keys include:

- `core_id`
- `type` (e.g., rhetorical, declarative)
- `level` (integer)
- `sweep` (optional; mostly for intervention data)

---

## 📁 Output Structure

Each basis is saved into a subfolder of your selected run's `basis/` directory.

- `.npz` contains the actual basis vectors (`basis_1`, `basis_2` or an ensemble matrix)
- `.json` contains metadata about how the basis was generated, including:
    - filters used
    - number of vectors found
    - CLI args
    - source run
    - timestamp

These files are automatically linked when you run SRM analysis later.

---

✅ After this, you're ready to run **SRM Analysis**, projecting either baseline or intervention vectors into your new basis.

---

## 🧭 Framework Note: Basis Compatibility

Projection basis files can be reused across multiple SRM runs—but only when key compatibility conditions are met. Misaligned bases may lead to meaningless projections, vector shape mismatches, or dropped data during analysis.

A basis is compatible with a given capture run if:

1. **Dimensionality matches**
   The original and target vector sets must have the same embedding size (e.g., 3072D for GPT-2 Small MLP activations). You cannot use a 768D residual-based basis with MLP vectors unless manually projected into that space.

2. **Model architecture and layer match**
   If a basis is generated from vectors captured at `blocks.11.mlp.hook_post`, it should only be applied to other vectors from the same model and layer. Projections across layers may behave inconsistently unless intentionally designed.

3. **Prompt metadata is compatible (for grouped SRM)**
   SRM itself does not require metadata for individual vector projection—but group-based plotting (e.g., by `type` or `level`) will silently ignore vectors if their metadata keys don't match the basis-linked expectations.

4. **Single-plane basis must be 2D**
   Only single-plane bases (i.e., with `basis_1` and `basis_2`) can be used for angular sweeping. Ensemble bases are not directly compatible with SRM projection unless reduced into a plane manually.

As a rule of thumb:

> *If two capture runs were generated using the same script configuration (layer, vector type, prompt structure), their basis and vectors are interoperable.*

## 🧭 Framework Note: Single Plane vs Ensemble

The SRM v6 Framework supports two distinct modes for generating basis vectors: `single_plane` and `ensemble`. While they use similar inputs, they serve different experimental goals and produce fundamentally different projection spaces.

**Single Plane Mode** is used for targeted projection. It constructs a 2D subspace—defined by two mean vectors—based on filtered subsets of captured activation data. While the filters may reflect semantic categories (e.g., rhetorical vs declarative), the key utility of this mode is its ability to explicitly define a projection plane aligned with a hypothesized axis of interest inside the model's representational space.

In our pilot experiments, for example, Neuron 373 was identified as a candidate modulator based on earlier ranking heuristics. It was paired with Neuron 2202 due to apparent co-behavior or complementary alignment. A projection plane was then constructed using vectors from prompt types that happened to load on those neurons, allowing the team to study activation trajectories, epistemic clustering, and angular divergence under modulation.

This approach allows a specific region of the embedding space to be isolated and studied directly. The method does not ask what planes exist in the data—it assumes one has already been identified, and instead asks: *how do representations behave within this plane?*

Technically:

- The projection space is a 2D plane formed by two mean vectors: `basis_1` and `basis_2`
- These vectors are calculated from filtered prompt groups
- This basis can be re-used across both baseline and intervention runs to ensure comparative integrity

**Ensemble Mode**, by contrast, is used for exploratory clustering. Instead of defining a plane, a metadata key (such as `type` or `level`) is provided. The system then groups vectors by that key, calculates a mean vector for each group, and stores these in an ensemble. The result is not a plane, but a structured set of vectors that can be used to study how representational space is organized around latent groupings.

This mode aligns with the original design intent of Bird SRM. It is well suited for mapping the angular distribution of prompt categories, identifying emergent structure, detecting symmetry or redundancy, and creating visualizations such as Compass Roses or angular heatmaps.

Unlike single plane mode, ensemble mode does not produce an explicit 2D bivector for rotation. While it is theoretically possible to approximate a 373–2202-aligned plane by filtering ensemble outputs, this is indirect and often lossy. Ensemble mode is not intended for hypothesis-driven projection, but for pattern discovery and cluster analysis.

Technically:

- The output is a matrix of group-averaged basis vectors keyed by label

- These can be used to analyze directional distributions, grouping behaviors, or conceptual overlaps

- Ensemble outputs are not directly suited to intervention comparisons without post-processing

The choice of mode reflects the underlying research question:

- Use **single plane** when the aim is to interrogate specific neuron axes, known contrasts, or intervention effects

- Use **ensemble** when the aim is to uncover latent group structure, visualize semantic resonance, or explore emergent geometry

Both modes are fully supported within the SRM v6 framework and may be used independently or in combination depending on the goals of a given experiment.

## 🧭 Framework Note: Identifying Neurons of Interest

Some parts of the experimental methodology underpinning this work are not detailed in this document. This is because their functionality is not yet included in v6. A future version of the framework—or a standalone tool—will include support for early-phase exploratory analysis, such as identifying candidate neurons for SRM interrogation. When that's available, an addendum will be added describing techniques like concept decomposition, correlation-based ranking, or embedding-aligned projection. This is often where the targeted SRM process begins: identifying where in the model geometry something *might* be happening.

## 🧭 Framework Note: Comparing Across Runs

Comparing directional resonance between baseline and intervention runs is a powerful application of the SRM v6 framework—particularly in its targeted experimental mode. It enables analysis of how specific neuron modulations reshape the internal geometry of representation space.

To compare two runs:

1. **Use the same projection basis** for both baseline and intervention analyses
   This ensures both sets of vectors are measured against a shared conceptual frame.

2. **Run SRM analysis independently** on each run using `analyze_srm_sweep.py`
   Each will produce angle-wise resonance data grouped by prompt metadata.

3. **Overlay or compare**
   After export to `.csv`, results can be:

   - Overlaid to visualize divergence
   - Subtracted angle-by-angle to highlight drift
   - Used to construct separate Compass Roses or rose-difference plots

This method surfaces:

- Angular realignment or disintegration under modulation
- Differential stability across epistemic groups
- Potential signatures of functional pressure or semantic collapse

This comparison workflow is not part of the original SRM method, but emerges naturally within this framework when paired with vector-level intervention.

# 🌀 Running SRM Analysis

Once a projection **basis** has been generated, SRM analysis can be run on any compatible set of captured vectors—baseline or intervened. The analysis projects each vector into the 2D plane, then rotates a probe vector through that space to measure alignment at each angle. This produces a resonance pattern showing which directions dominate, cluster, or diverge.

This is the step where geometric structure becomes visible.

---

## 🔧 Requirements

To run SRM analysis, you'll need:

- A completed capture run (baseline or intervened)
- A basis file (`.npz`) created using `generate_basis_vectors.py`
- Matching dimensionality (3072D if using GPT-2 Small)
- An output folder to store plots, CSVs, and metadata

If using the default script, all required files will be selected interactively.

---

## ▶️ Command Format

```
python analyze_srm_sweep.py
```

You will be prompted to:

1. Select a run folder (from the `experiments/` directory)
2. Choose a vector file (typically from `capture/vectors/`)
3. Choose a basis file (typically from `basis/`)
4. Optionally set parameters (cone epsilon, number of angles, vector norm behavior)

---

## ⚙️ What It Does

- Loads the selected vector set (baseline or intervention)
- Projects each high-dimensional vector into the 2D subspace defined by the basis
- Normalizes the projected vectors
- Defines a rotating **probe vector** that sweeps through the plane in uniform angular steps (e.g. 180 steps from 0 to 2π radians)
- For each angle, measures how many projected vectors fall within a directional cone of width `epsilon` around the current probe direction
- Aggregates results into:
  - **mean resonance** (average similarity across all vectors per angle)
  - **grouped hit counts** (if vectors are tagged with metadata like `type`, `level`, etc.)
  - **full trace data** for visualization or custom metric calculations

---

## 📁 Output Files

Analysis results are saved in:

```
experiments/
└── run_baseline_.../
    └── analyses/
        └── srm_grouped_by_level_basis_N373-N2202/
            ├── srm_plot_mean_sim_only.png
            ├── srm_plot_thresholded_count.png
            ├── srm_data_grouped_by_level_basis_N373-N2202.csv
            ├── meta_srm_grouped_by_level_basis_N373-N2202.json
            └── group_metadata.json
```

Includes:

- **Plots** (PNG):
  - Mean similarity per angle
  - Thresholded hit counts by group (e.g. epistemic type)
- **Data CSV**:
  - Full angular data for each group, useful for further analysis. “Full angular data per group” includes cosine similarity at each angular bin, and can be used to reconstruct or extend Compass Rose plots (see: Compass Rose).
- **Metadata**:
  - Logs all config settings, basis file used, vector source, sweep parameters

---

## 🧠 SRM Mechanics (Simplified)

Each vector is projected into a 2D plane, normalized, and compared to a rotating probe vector at regular angular steps. The probe defines a directional cone (`ε`) around each angle. When a vector falls within the cone, it’s counted or scored.

This method reveals:

- Dominant alignment directions
- Rotational clustering by group
- Effects of neuron modulation (by comparing intervention vs baseline)

---

✅ Once this step is complete, results can be interpreted visually, exported, or used as the input to higher-level visualizations (e.g. Compass Roses, angular drift maps).

# 📝 Creating Prompt Sets

Prompt sets are the input data used in SRM experiments. They define what is shown to the model, how activations are grouped, and what metadata is available for analysis. In SRM v6, prompt sets are stored in `.txt` format and follow a specific block structure.

Each block defines a **fixed semantic core**—a narrative idea or proposition—and then varies the **epistemic framing** and **certainty level** across multiple prompt variants.

This design ensures that meaning is held relatively constant while epistemic stance is varied—ideal for experiments measuring alignment, drift, or angular clustering.

---

## 📄 File Structure

Each prompt block contains:

- A **CORE_ID** marker
- A natural language **PROPOSITION** statement (for human traceability)
- Five numbered levels (`[LEVEL 1]` to `[LEVEL 5]`). 1 represents the weakest epistemic certainty.
- Within each level, one prompt per **epistemic type**:
    - `observational`
    - `declarative`
    - `authoritative`
    - `rhetorical`
      *(Other types can be added, as detailed below, but these are assumed by default)*

Prompts are written in plain text. No quotes, no JSON. Just clean spacing.

---

## ✅ Example Block

```
>> CORE_ID: red_light_warning

>> PROPOSITION: The red warning light is flashing.


[LEVEL 3]

observational: The red warning light is flashing.

declarative: The system issued a red-light alert.

authoritative: System logs confirm red alert status.

rhetorical: Red light flashes. The moment begins.
```

This block defines a single narrative moment and its four epistemic framings at **Level 3** certainty.

---

### 🔍 How Prompts Are Parsed

The system automatically parses:

- `CORE_ID` from the header
- `type` from the prefix of each line (e.g., `observational:`)
- `level` from the surrounding `[LEVEL N]` bracket
- The text that follows the colon as the actual prompt

Each unique combination (CORE_ID × TYPE × LEVEL) becomes its own prompt and vector key in the format:

```
core_id=red_light_warning type=declarative level=3
```

This metadata is embedded automatically in all downstream vector keys, basis filters, logs, and plots.

---

## 🧠 Design Philosophy

This format is designed to:

- Enforce **semantic consistency across variants**
- Isolate the effects of epistemic stance and certainty, across and within groups.
- Enable easy grouping for SRM comparison (e.g., “all level 3 rhetorical prompts”)

It also ensures that when analyzing angular alignment or drift, **changes are epistemic, not conceptual**.

You might benefit from reflection on the difference here: between knowing/knowledge and ideas/concepts. Subtle, but centrally important to this framework. We are not (by design or default) measuring “concept neurons” nor interested in “concepts” per se, but in epistemic function, such as levels of certainty and their topological effects.

---

## 💡 Prompt Writing Tips

*This is for default experimentation - you can break these rules if you have a good reason, of course.*

- Keep `PROPOSITION` simple and concrete—avoid abstract phrasing
- Each level should reflect increasing **epistemic certainty**
- Prompt tone should vary only in framing, not in fact pattern
- Use consistent epistemic types across all cores if possible (missing types may break group alignment in ensemble mode)

# 📝Redesigning prompt sets:

## ✅ Yes — Additional Epistemic Types Are Supported

The framework doesn't hardcode the four types (`observational`, `declarative`, `authoritative`, `rhetorical`) anywhere. Instead:

It parses the type name **dynamically** from each prompt line using this pattern:
`epistemic_type: prompt text`

Whatever comes before the first colon (`:`) on a line becomes the `type` key. The parser assumes well-formed lines and doesn't validate unknown types. If the type includes spaces or colons, it may break parsing silently.

This key is then embedded into the vector metadata like:
`core_id=presence_by_door type=speculative level=2`

So if you add a new type—say, `speculative:` or `interrogative:`—the system will simply treat it as a new category. Hopefully.

---

## ⚠️ But! Grouping Depends on Coverage

If you introduce a new epistemic type but **don't include it for every CORE_ID × LEVEL**, a few things may happen:

1. **Basis generation (ensemble mode)** may produce uneven groups
   If you use `--ensemble_group_key type` and one type only appears in a few rows, it'll still create a mean vector—but that group may be noisy or statistically weak.

2. **SRM analysis grouped by `type`** will include the new type, but if your analysis relies on balanced comparisons, that asymmetry could skew interpretation.

3. **Plots** will show the new type automatically, but may look odd if it only appears at one level (e.g. "interrogative" shows up for level 4, but nowhere else).

---

## 🧪 Recommendation for Adding Types

If you're going to experiment with new types (which is awesome), it's best to:

- Add them **consistently across all cores and levels**
- Maintain the naming style (`type_name:` with no caps or spaces)
- Consider running an ensemble basis to see how the new type clusters

# 📂 File & Folder Structure

The SRM v6 Framework organizes every experiment into a **self-contained run directory**, stored under a common `experiments/` folder. Each run—whether baseline or intervention—has its own subdirectory, with clearly separated outputs for capture, basis vectors, and analysis results.

This structure is designed to support reproducibility, script automation, and manual inspection.

---

## 🧬 Top-Level Layout

```
experiments/

├── run_baseline_L11N373_gridA_20250418_113231/

│   ├── capture/

│   ├── basis/

│   └── analyses/

└── run_intervened_L11N373_gridA_20250418_115804/

    ├── capture/

    ├── basis/

    └── analyses/
```

Each run is timestamped and labeled with basic metadata: run type (baseline/intervened), layer, neuron, grid name, and date. These are auto-generated but can be manually renamed if needed.

---

## 📁 capture/

This folder holds all raw outputs from the vector capture phase:

```
capture/
├── vectors/
│   └── captured_vectors_[baseline|intervened].npz
├── metadata/
│   └── run_metadata_[baseline|intervened].json
└── logs/
    └── run_log_[baseline|intervened].md
```

- **Vectors**: `.npz` file storing all captured activation vectors, one per prompt
- **Metadata**: settings used during capture (layer, sweep values, prompt file, etc.)
- **Log**: human-readable log of prompt completions, warnings, and notes

---

### 📁 basis/

Created during basis generation using `generate_basis_vectors.py`.

```
basis/
├── basis_single_plane_t_decl_vs_t_rhet.npz
├── basis_single_plane_t_decl_vs_t_rhet.json
├── basis_ensemble_by_type.npz
└── basis_ensemble_by_type.json
```

- **Single plane** files contain `basis_1` and `basis_2` vectors
- **Ensemble** files contain a matrix of basis vectors (one per group key)
- **Metadata JSONs** record filters used, group counts, and generation method

These files are linked automatically by the SRM analysis script.

---

## 📁 analyses/

Each analysis run generates a new subfolder based on vector source and basis used.

```
analyses/
└── srm_grouped_by_level_basis_N373-N2202/
    ├── srm_data_grouped_by_level.csv
    ├── srm_plot_mean_sim_only.png
    ├── srm_plot_thresholded_count.png
    ├── meta_srm_grouped_by_level.json
    └── group_metadata.json
```

- **Plots**: show angular similarity or hit counts per group
- **CSV**: contains raw resonance data, grouped by key
- **Metadata**: records sweep config, cone epsilon, angle steps, linked basis, etc.

These folders can be safely duplicated, moved, or shared—each one is self-contained.

---

## 🔁 Interoperability

- Scripts automatically link each step to prior outputs (e.g., analysis selects basis from same run)
- File and folder naming is consistent and predictable for easy scripting
- You can run multiple analyses, sweeps, or basis generations per run folder—results will not be overwritten

## 🧭 Compass Rose Visualizations

Compass Rose plots are radial, polar-style histograms that visualize how different prompt groups—typically defined by `epistemic type` or `certainty level`—distribute their directional resonance in a shared SRM projection plane.

Unlike linear similarity plots, the Compass Rose renders *angular semantic pressure* as shape. Each "petal" represents directional resonance: where groups cluster, align, or diverge in a latent plane such as 373–2202.

---

### 🔧 What It Requires

To generate a Compass Rose, you'll need:

- A `.npz` vector file (baseline or intervened, comparing *both* can be instructive)
- Structured keys with embedded metadata (`core_id`, `type`, `level`)
- An SRM basis file (single plane only)
- A cosine sweep function across the basis plane (see `analyze_srm_sweep.py`)
- A plotting pipeline that:
    - Bins vectors into angular slices
    - Groups by metadata (e.g., `type`)
    - Averages cosine similarity per group per angle
    - Renders stacked bars in polar coordinates

This is a **secondary visualization step**—data is presumably first generated by running SRM analysis and saved to `.csv`.

---

### 🎯 How It Works

1. **Projection**
   Vectors are projected into a 2D basis (e.g. 373–2202) and normalized.

2. **Sweep & Spotlight**
   A rotating probe vector is swept across 360°, typically in 5° steps (72 total). At each angle, cosine similarity is computed between each projected vector and the spotlight direction.

3. **Angular Binning**
   Vectors are binned into angular slices. For each group (e.g., rhetorical), similarity scores are aggregated per bin.

4. **Bar Height Calculation**
   The bar for each group in a bin reflects its **mean cosine similarity** to the spotlight vector. This reflects *resonance*, not frequency.

5. **Stacked Polar Rendering**
   Final bars are rendered using `matplotlib.pyplot.bar()` in polar mode, stacked per group to show angular overlap and contrast.

---

## 🧠 Why It Matters

The Compass Rose makes directional structure visible. Unlike bar or line plots, which track similarity over time or conditions, this chart reveals:

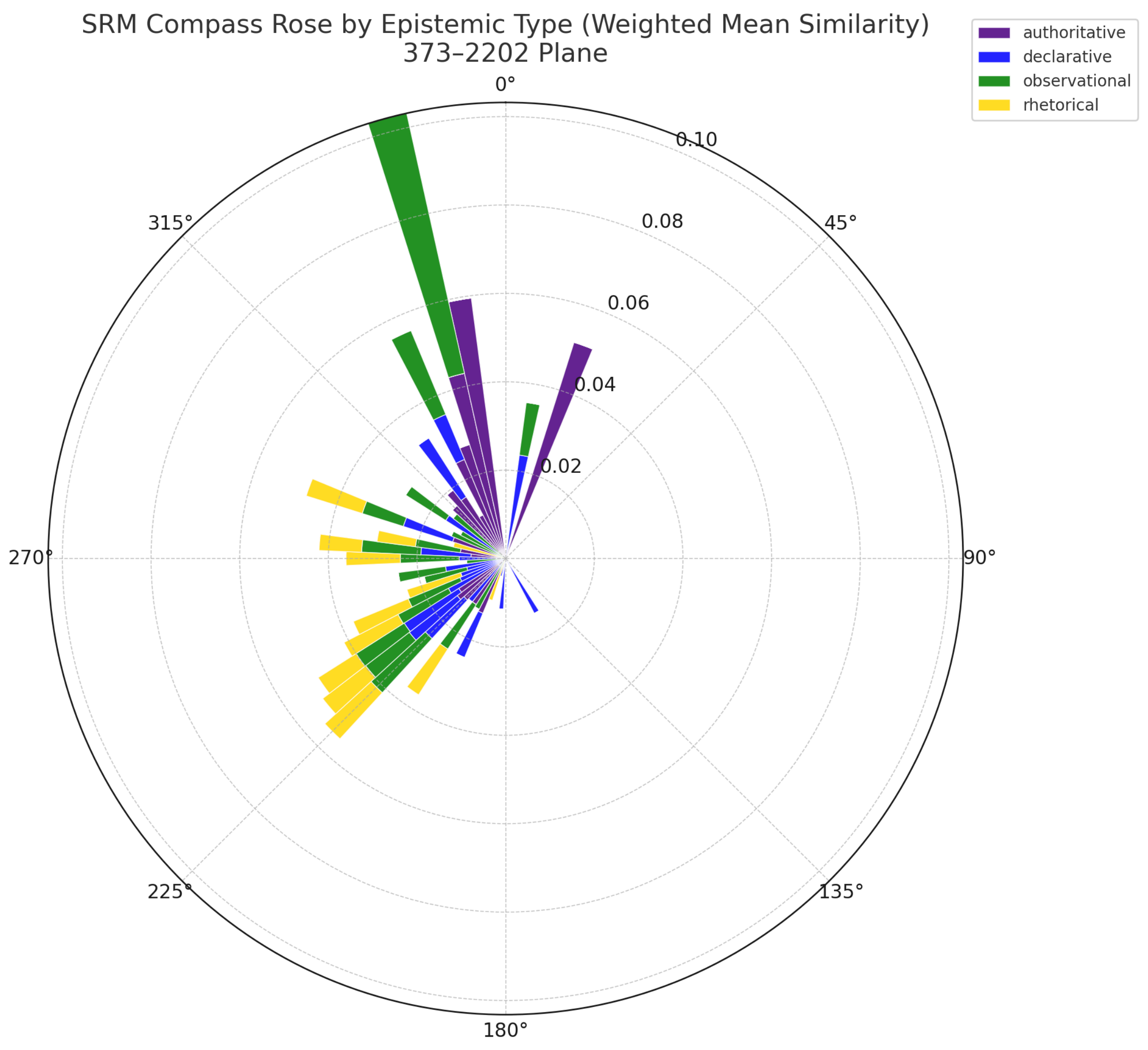
- Alignment clusters (e.g. rhetorical prompts converging at ~250°)
- Rotational divergence across types (e.g. observational vs authoritative)
- Group-level angular spread or tightness
- Latent directional geometry of meaning

It's particularly useful for identifying **semantic attractors** in neuron-defined planes.

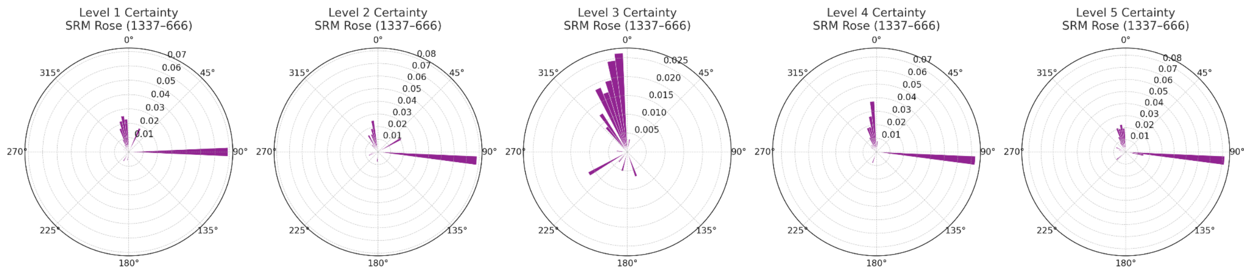
---

## 🧪 Output Examples

**All-Type Rose**: Shows how each group (e.g. "authoritative") spreads directionally across the plane according to weighted mean similarity. Good for overviews of general structure across the dataset.

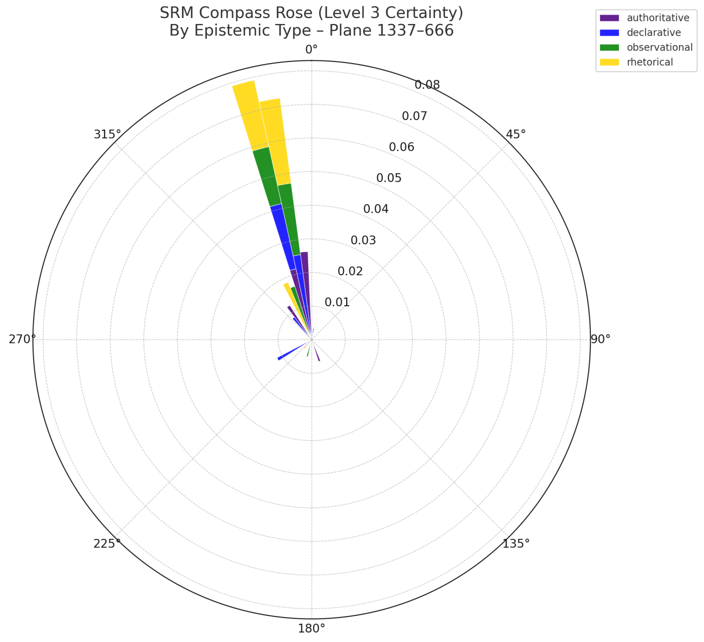


**Grid of Roses**: Used to track divergence across levels (e.g. one rose per `[LEVEL N]`).



*Note how at certainty level 3 something shifts the entire topology. Is this a phase transition or similar? Something unusual about the prompt sets at L3? Or is it something else? These are the curiosities the framework helps surface.*

**Stacked-Type Rose**: Shows all types layered together for a single certainty level or core. Building on the last stack of 5, we drill further into 3's own individual topology at that level, like so:



*Note how each prompt's epistemic type has notable topological effects, showing different levels of resonance. Note also, how they are structured **intuitively** along the same gradients as their strength: authoritative, the strongest epistemic type, then declarative, and so on. What is this?*

These can be rendered with transparency overlays, color-coding, or filtered subsets (e.g. only `core_id=threat_warning`).

## ⚠️ Potential Pitfalls

- **Spikiness** is not a rendering bug. It usually reflects sparse but strongly aligned vectors in low-density bins.
- If few prompts land in a bin, a single vector with high alignment can create a tall bar.
- For smoother visuals, try:
    - Count-based binning instead of mean similarity
    - Gaussian kernel smoothing across angle bins
    - Normalizing each group to its own max (relative resonance)

---

## 🧰 Prompting GPT to Generate One

To regenerate or modify a Compass Rose using GPT (as we did in development), prompt with:

> *“Using cosine similarity from projected 3072D baseline vectors into the 373–2202 basis plane, bin by angle (72 bins, 5° each), group by* `type`*, and plot stacked mean similarities in polar coordinates. Use bar height = average cosine per type per bin.”*

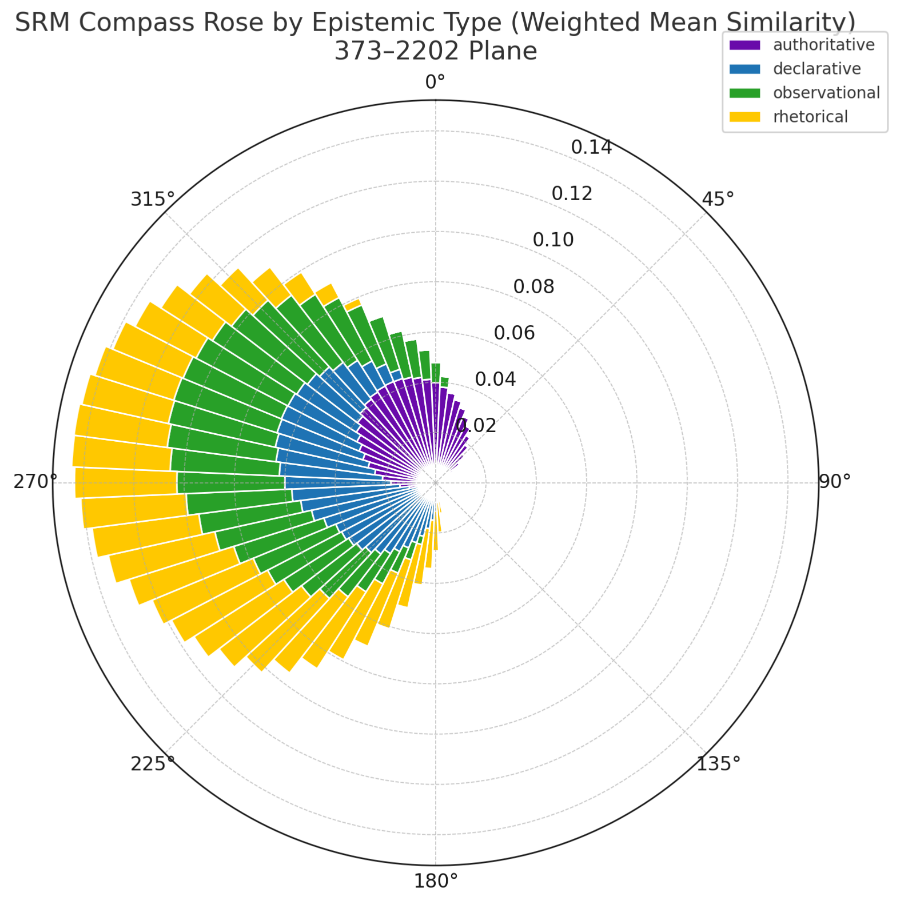
You will need to provide the data sets. For a more comprehensive understanding, you can feed GPT the python code, as well as this documentation. A full .zip file of the entire project can also be given to an LLM for parsing. See: Download Parsable Project.

You can request:

- Count-based variant
- Per-type normalization
- Overlay comparisons
- Certainty-level subplots (rose grids)
- Residual-stream vs MLP comparisons

---

## ✅ Why v6 Looked Spikier Than v5



*An earlier version of the experiment yielded this result. Understanding the differences may be instructive:*

Both versions used the same prompt input and methodology. The difference came from:

- Vector source: residual vs MLP (v6 used richer 3072D MLP vectors)
- Sampling fidelity: v6 kept all types, even sparse bins
- Plotting: v6 used literal mean similarity, revealing sharper semantic pulls

**The jaggedness in v6 is a feature**—it reveals real directional sparsity and epistemic clustering. The smoother rose may have masked this with interpolation or under-sampled types.

In short, your bar *probably* should look like spiky clock hands, not clamshells (how beautiful is that phyllotaxis though!).