

The Specification Dilemma

Experiment Specification and Scaffolding

April 2026

Abstract

This document specifies a small empirical probe for the claim that as specification sparsity increases, pairwise semantic similarity across outputs generated from plausibly-varied user prompts increases. The experiment is designed to run locally against a strong open-weights model served by LMStudio, using sentence embeddings to measure output homogeneity in a sparse-vs-dense matched-pairs design, where 30 imagined users with distinct underlying intents each contribute both a sparse and a dense prompt. The document includes the full specification, Python scaffolding for generation and analysis, prompt files, and a project layout ready to execute.

Contents

1 Hypothesis	2
2 Design	2
2.1 Task selection	2
2.2 Sample size	2
2.3 Prompt generation	2
2.4 Generation parameters	3
2.5 Similarity measurement	3
2.6 Statistical analysis	3
2.7 Expected result and falsification criteria	3
2.8 Optional robustness checks	4
3 Project Layout	4
4 Configuration	5
4.1 config.yaml	5
4.2 requirements.txt	5
5 Prompt Files	5
5.1 prompts/sparse.json	5
5.2 prompts/dense.json (example entry; see repo for all 30)	6
6 Generation Script	8

6.1	<code>generate.py</code>	8
7	Embedding Script	10
7.1	<code>embed.py</code>	10
8	Similarity Computation	12
8.1	<code>similarity.py</code>	12
9	Statistical Analysis	14
9.1	<code>stats.py</code>	14
10	Plotting	17
10.1	<code>plot.py</code>	17
11	Pipeline Orchestrator	18
11.1	<code>run_all.py</code>	18
12	README	19
12.1	<code>README.md</code>	19
13	Essay Integration Notes	20

1 Hypothesis

As specification sparsity increases (fewer tokens / less detail in the prompt), pairwise semantic similarity across outputs generated from plausibly-varied user prompts increases. Equivalently: sparse specification produces homogenized output across users, even when those users phrase their requests differently.

This probes the core empirical claim of the essay “The Specification Dilemma”: that the mechanism by which inference-heavy collaboration produces homogeneity is the convergence of outputs onto the model’s shared priors when human specification is insufficient to push the model off its modes.

2 Design

A two-condition design with **matched pairs at the user level**, comparing output similarity distributions in a **sparse specification** condition versus a **dense specification** condition.

The unit of variation is *the user*. Each condition contains N prompts, and the two conditions share the same N imagined users. Each user has a fixed underlying intent (audience, thesis, tone, voice, opening move, structural constraint), and that intent is expressed twice: once as a sparse prompt (topic only, in the user’s natural register) and once as a dense prompt (the intent in full). The sparse condition simulates those 30 users when they vibe; the dense condition simulates the same 30 users when they specify. Controlling the underlying intents across conditions tightens the contrast: any divergence between the two similarity distributions is attributable to specification completeness rather than to differences between the populations of users sampled.

2.1 Task selection

One task: “**Write the opening 300 words of a blog post about remote work.**” Rationale: a task many users actually perform with LLMs, enough creative latitude for homogenization to matter, output length tractable for embedding-based similarity.

2.2 Sample size

$N = 30$ per condition (60 prompts total). This yields $\binom{30}{2} = 435$ pairwise comparisons per condition, which is enough for a reasonably tight confidence interval on mean similarity.

2.3 Prompt generation

Procedure. First, write $N = 30$ underlying intents — one per imagined user, spanning distinct audiences, theses, tones, voices, opening moves, and structural constraints. Then, for each intent, produce a matched pair of prompts:

- **Sparse prompt** (5–20 tokens, one sentence): what that user would type when underspecifying — topic only, in their natural register (formal vs. casual, polite vs. terse, with or without a length cue). The sparse prompt must carry no audience, thesis, tone, voice, or structural specification. Smuggling those in would collapse the contrast the experiment is trying to measure.
- **Dense prompt** (150–300 tokens): the intent in full — target audience, thesis, tone, structural choices, vocabulary register, things to avoid, and author voice.

The two prompt files preserve order: `sparse.json[i]` and `dense.json[i]` are the same imagined

user’s two expressions. Different imagined users should specify *different* target audiences, tones, and angles, so the 30 dense prompts span the space of plausible divergent intents.

Pre-registration discipline: write and freeze all 30 intents and all 60 prompts before running any generation. Do not iterate on prompts after seeing outputs.

2.4 Generation parameters

- **Model:** one strong open-weights instruction-tuned model on LMStudio (e.g. Qwen2.5-72B-Instruct or Llama-3.3-70B-Instruct), full precision or Q8. Held constant across all 60 generations.
- **Temperature:** 0.7
- **Top-p:** 0.95
- **Max tokens:** 500
- **Seed:** per-prompt deterministic seed (`seed = prompt_index`) for reproducibility
- **Generations per prompt:** 1

2.5 Similarity measurement

Embedding model: sentence-transformers/all-mpnet-base-v2 as the default; BAAI/bge-large-en-v1.5 as an optional robustness check.

Metric: cosine similarity between embedding vectors of full 300-word outputs.

Aggregation: for each condition, compute all 435 pairwise cosine similarities, producing two distributions of similarity scores.

2.6 Statistical analysis

Primary test: two-sample t-test (or Mann–Whitney U if distributions are non-normal) comparing sparse and dense similarity distributions.

Reported statistics: mean similarity per condition, standard deviation, test statistic, p -value, and Cohen’s d .

Secondary visualization: overlaid violin plots of the two similarity distributions. More rhetorically useful in the essay than the p -value alone.

Dependence caveat: the 435 pairwise similarities per condition are not independent (each output appears in 29 pairs), so the effective sample size is smaller than 435. A bootstrap at the output level (resample outputs with replacement, recompute mean pairwise similarity, repeat 10,000 times) gives a more conservative interval. Both the naive t-test and the bootstrap CI should be reported; the scaffolding below computes both.

2.7 Expected result and falsification criteria

If the hypothesis holds: sparse-condition mean pairwise similarity is meaningfully higher than dense-condition mean similarity, with large effect size (Cohen’s $d > 0.8$) and $p < 0.01$.

What would weaken or falsify it:

- Similar means across conditions (no homogenization effect).

- Small effect size even if statistically significant.
- Dense condition *higher* than sparse — would be surprising and worth investigating (possibly an artifact of shared specification language leaking into outputs).

2.8 Optional robustness checks

In priority order, if time permits:

1. Re-run with a second task to check task-generalizability.
2. Re-run with a different embedding model to check metric-robustness.
3. Re-run with a different base LLM to check model-generalizability.

3 Project Layout

```

1 experiment/
2   prompts/
3     sparse.json # 30 sparse prompts, matched to dense by index
4     dense.json # 30 dense prompts, matched to sparse by index
5   outputs/
6     sparse/ # 30 .txt files, one per generation
7     dense/ # 30 .txt files, one per generation
8   embeddings/
9     sparse.npy # (30, D) embedding matrix
10    dense.npy # (30, D) embedding matrix
11  results/
12    pairwise.csv # all pairwise similarities, condition labeled
13    stats.json # test statistics and summary metrics
14    plot.png # violin + histogram comparison
15  config.yaml # model name, temperature, paths
16  smoke_test.py # pre-flight: connectivity, seed, latency
17  generate.py # runs LMStudio generations
18  embed.py # computes sentence embeddings
19  similarity.py # computes pairwise cosine similarities
20  stats.py # t-test, Mann-Whitney, bootstrap, effect size
21  plot.py # violin plot of similarity distributions
22  run_all.py # orchestrator: runs the full pipeline
23  pyproject.toml # uv-managed environment
24  requirements.txt # pip fallback
25  README.md

```

Dependencies. openai (LMStudio client), sentence-transformers, numpy, scipy, matplotlib, seaborn, pandas, pyyaml, tqdm.

Runtime estimate. Generation: 60 prompts \times \sim 15–30s each \approx 15–30 min on strong hardware. Embedding + analysis + plotting: $<$ 1 min. Total wall time: well under an hour once prompts are frozen.

4 Configuration

4.1 config.yaml

```

1 # LMStudio server settings
2 lmstudio:
3   base_url: "http://localhost:1234/v1"
4   api_key: "lm-studio" # placeholder; LMStudio ignores this
5   model: "qwen2.5-72b-instruct" # name as it appears in LMStudio
6
7 # Generation parameters
8 generation:
9   temperature: 0.7
10  top_p: 0.95
11  max_tokens: 500
12
13 # Embedding model
14 embedding:
15   model: "sentence-transformers/all-mpnet-base-v2"
16
17 # Paths (relative to project root)
18 paths:
19   prompts_dir: "prompts"
20   outputs_dir: "outputs"
21   embeddings_dir: "embeddings"
22   results_dir: "results"
23
24 # Analysis
25 analysis:
26   bootstrap_iterations: 10000
27   random_seed: 42

```

4.2 requirements.txt

```

1 openai≥1.40.0
2 sentence-transformers≥3.0.0
3 numpy≥1.26.0
4 scipy≥1.13.0
5 pandas≥2.2.0
6 matplotlib≥3.8.0
7 seaborn≥0.13.0
8 pyyaml≥6.0
9 tqdm≥4.66.0

```

5 Prompt Files

The 30 sparse prompts below are the frozen set shipped with this experiment. `sparse.json[i]` and `dense.json[i]` correspond to the same imagined user: the dense prompt expresses that user's full intent, and the sparse prompt is what the same user would type when underspecifying. The sparse set below carries no audience, thesis, tone, voice, or structural direction; only register, punctuation, and length cues differ across users. Freeze your final set before running any generations.

5.1 prompts/sparse.json

```

1 [

```

```

2  "remote work blog post intro, 300 words",
3  "write the opening of a blog post on remote work",
4  "Draft an opening for a blog post about remote work.",
5  "can you write a blog post intro about remote work",
6  "blog intro about working from home, 300 words",
7  "need a remote work blog intro, ~300 words",
8  "Please draft the opening 300 words of a blog post about remote work.",
9  "I'd like the opening of a blog post about remote work, around 300 words.",
10 "remote work blog post opener, 300 words",
11 "blog intro, remote work, 300 words",
12 "first 300 words of a blog post on remote work",
13 "opening 300 words, blog post about remote work",
14 "hey can you write a remote work blog intro",
15 "write the first 300 words of a blog on remote work",
16 "Draft the opening paragraph of a blog post on remote work.",
17 "Please write the opening of a blog post about remote work, approximately 300 words.",
18 "Write the opening of a blog post about remote work, around 300 words.",
19 "blog post, remote work, first 300 words",
20 "could you write the intro to a blog post on remote work",
21 "Write the opening 300 words of a blog post about remote work.",
22 "opening for a blog post on remote work, ~300 words",
23 "write a blog intro about remote work please",
24 "Please draft the opening of a blog post about remote work.",
25 "blog post opening on remote work, about 300 words",
26 "Draft opening 300 words --- blog post, remote work.",
27 "write me a blog intro about remote work",
28 "Please produce the opening of a blog post on remote work, approximately 300 words.",
29 "can u write a remote work blog intro",
30 "need the opener for a blog post about remote work, ~300 words",
31 "Write the opening of a blog post about remote work."
32 ]

```

5.2 prompts/dense.json (example entry; see repo for all 30)

The dense prompts should each push the model in a different direction. The first entry, shown below, pairs with the first sparse prompt above and illustrates the shape:

```

1  [
2  "Write the opening 300 words of a blog post about remote work. The target audience is mid-career
   software engineers at 50-500 person startups who have worked remotely for 3+ years and are
   tired of both 'remote work is utopia' and 'return to office' takes. The tone should be dry,
   slightly weary, and specific rather than abstract. Open with a concrete observation about a
   small, texture-of-daily-life detail rather than a statistic or rhetorical question. Avoid
   the words 'unprecedented', 'new normal', 'journey', 'landscape', and 'game-changer'. The
   thesis should be that remote work's real cost is not productivity or culture but the erosion
   of ambient professional development - the kind of learning that happens when you overhear a
   senior engineer debug something. Voice should resemble a technical writer who also reads
   literary essays. No bullet points. End the opening paragraph on a sentence that turns the
   observation into a question the rest of the post will answer."
3  // ... 29 more, each specifying a different audience, thesis, tone, and
4  // structural constraint. See guidance below.
5  ]

```

Guidance for producing the 30 intents and matched prompts. Vary each of these axes independently across the 30 intents: target audience (engineers, designers, HR leaders, small-business owners, parents, new grads, academics, tradespeople, freelancers, founders, managers,

journalists, lawyers, clinicians, etc.); thesis (productivity, loneliness, asynchrony, hiring, real estate, inequality, rituals, surveillance, accessibility, capex amortization, etc.); tone (dry, earnest, skeptical, celebratory, investigative, personal-essay, confessional, technocratic, etc.); opening move (anecdote, statistic-that-turns, quote, scene, counter-intuitive claim, literary reference, quoted memo, etc.); voice (technical writer, essayist, journalist, first-person blogger, trade-publication editor, policy analyst, etc.); structural constraint (no bullets, one-sentence paragraphs, tight lede, delayed thesis, circular, etc.). For each intent, write the dense prompt that expresses it fully; then write the sparse prompt as what that same user would type when underspecifying — topic-only, in their natural register, carrying none of the specification content. The goal is that if the mechanism holds, the sparse-condition outputs will converge despite 30 different underlying intents, while the dense-condition outputs will diverge because the specifications actually push the model toward different regions of output space.

6 Generation Script

6.1 generate.py

```

1  """Generate completions for sparse and dense prompts via LMStudio.
2
3  LMStudio exposes an OpenAI-compatible server (default: localhost:1234).
4  Start the server from LMStudio's "Local Server" tab before running.
5  """
6  from __future__ import annotations
7
8  import json
9  import os
10 from pathlib import Path
11
12 import yaml
13 from openai import OpenAI
14 from tqdm import tqdm
15
16
17 def load_config(path: str = "config.yaml") →dict:
18     with open(path, "r") as f:
19         return yaml.safe_load(f)
20
21
22 def make_client(cfg: dict) →OpenAI:
23     return OpenAI(
24         base_url=cfg["lmstudio"]["base_url"],
25         api_key=cfg["lmstudio"]["api_key"],
26     )
27
28
29 def generate_one(
30     client: OpenAI,
31     model: str,
32     prompt: str,
33     temperature: float,
34     top_p: float,
35     max_tokens: int,
36     seed: int,
37 ) →str:
38     """Single completion. Returns the assistant message content."""
39     response = client.chat.completions.create(
40         model=model,
41         messages=[{"role": "user", "content": prompt}],
42         temperature=temperature,
43         top_p=top_p,
44         max_tokens=max_tokens,
45         seed=seed,
46     )
47     return response.choices[0].message.content or ""
48
49
50 def run_condition(
51     client: OpenAI,
52     cfg: dict,
53     condition: str,
54 ) →None:
55     prompts_path = Path(cfg["paths"]["prompts_dir"]) / f"{condition}.json"

```

```
56 outputs_dir = Path(cfg["paths"]["outputs_dir"]) / condition
57 outputs_dir.mkdir(parents=True, exist_ok=True)
58
59 with open(prompts_path, "r") as f:
60     prompts = json.load(f)
61
62 gen_cfg = cfg["generation"]
63 model = cfg["lmstudio"]["model"]
64
65 for i, prompt in enumerate(tqdm(prompts, desc=f"{condition}")):
66     out_file = outputs_dir / f"{i:02d}.txt"
67     if out_file.exists():
68         continue # resume support
69     text = generate_one(
70         client=client,
71         model=model,
72         prompt=prompt,
73         temperature=gen_cfg["temperature"],
74         top_p=gen_cfg["top_p"],
75         max_tokens=gen_cfg["max_tokens"],
76         seed=i,
77     )
78     out_file.write_text(text, encoding="utf-8")
79
80
81 def main() ->None:
82     cfg = load_config()
83     client = make_client(cfg)
84     for condition in ("sparse", "dense"):
85         run_condition(client, cfg, condition)
86     print("Generation complete.")
87
88
89 if __name__ == "__main__":
90     main()
```

7 Embedding Script

7.1 embed.py

```

1  """Compute sentence embeddings for each generation in each condition."""
2  from __future__ import annotations
3
4  from pathlib import Path
5
6  import numpy as np
7  import yaml
8  from sentence_transformers import SentenceTransformer
9  from tqdm import tqdm
10
11
12  def load_config(path: str = "config.yaml") →dict:
13      with open(path, "r") as f:
14          return yaml.safe_load(f)
15
16
17  def load_outputs(outputs_dir: Path) →list[str]:
18      """Load all .txt outputs from a condition directory, sorted by filename."""
19      files = sorted(outputs_dir.glob("*.txt"))
20      return [f.read_text(encoding="utf-8") for f in files]
21
22
23  def embed_condition(
24      model: SentenceTransformer,
25      texts: list[str],
26  ) →np.ndarray:
27      """Return (N, D) embedding matrix. L2-normalized for cosine similarity."""
28      embeddings = model.encode(
29          texts,
30          batch_size=8,
31          show_progress_bar=True,
32          convert_to_numpy=True,
33          normalize_embeddings=True,
34      )
35      return embeddings
36
37
38  def main() →None:
39      cfg = load_config()
40      model = SentenceTransformer(cfg["embedding"]["model"])
41
42      outputs_root = Path(cfg["paths"]["outputs_dir"])
43      emb_root = Path(cfg["paths"]["embeddings_dir"])
44      emb_root.mkdir(parents=True, exist_ok=True)
45
46      for condition in ("sparse", "dense"):
47          texts = load_outputs(outputs_root / condition)
48          if not texts:
49              print(f"No outputs found for {condition}; skipping.")
50              continue
51          print(f"Embedding {len(texts)} {condition} outputs...")
52          embeddings = embed_condition(model, texts)
53          np.save(emb_root / f"{condition}.npy", embeddings)
54          print(f"Saved {condition}.npy with shape {embeddings.shape}")
55

```

```
56  
57 if __name__ == "__main__":  
58     main()
```

8 Similarity Computation

8.1 similarity.py

```

1  """Compute pairwise cosine similarities within each condition."""
2  from __future__ import annotations
3
4  from itertools import combinations
5  from pathlib import Path
6
7  import numpy as np
8  import pandas as pd
9  import yaml
10
11
12  def load_config(path: str = "config.yaml") →dict:
13      with open(path, "r") as f:
14          return yaml.safe_load(f)
15
16
17  def pairwise_cosine(embeddings: np.ndarray) →tuple[np.ndarray, list[tuple[int, int]]]:
18      """Return (similarities, index_pairs) for all i<j pairs.
19
20      Assumes embeddings are L2-normalized, so cosine = dot product.
21      """
22      n = embeddings.shape[0]
23      pairs = list(combinations(range(n), 2))
24      sims = np.array([
25          float(embeddings[i] @ embeddings[j]) for i, j in pairs
26      ])
27      return sims, pairs
28
29
30  def main() →None:
31      cfg = load_config()
32      emb_root = Path(cfg["paths"]["embeddings_dir"])
33      results_root = Path(cfg["paths"]["results_dir"])
34      results_root.mkdir(parents=True, exist_ok=True)
35
36      rows = []
37      for condition in ("sparse", "dense"):
38          emb_path = emb_root / f"{condition}.npy"
39          if not emb_path.exists():
40              print(f"Missing embeddings for {condition}; skipping.")
41              continue
42          embeddings = np.load(emb_path)
43          sims, pairs = pairwise_cosine(embeddings)
44          for (i, j), s in zip(pairs, sims):
45              rows.append({
46                  "condition": condition,
47                  "i": i,
48                  "j": j,
49                  "cosine": s,
50              })
51          print(
52              f"{condition}: n_outputs={embeddings.shape[0]}, "
53              f"n_pairs={len(sims)}, mean={sims.mean():.4f}, "
54              f"std={sims.std(ddof=1):.4f}"
55          )

```

```
56
57     df = pd.DataFrame(rows)
58     df.to_csv(results_root / "pairwise.csv", index=False)
59     print(f"Saved {results_root / 'pairwise.csv'}")
60
61
62 if __name__ == "__main__":
63     main()
```

9 Statistical Analysis

9.1 stats.py

```

1  """Statistical tests and summary metrics for the similarity comparison.
2
3  Reports:
4  - Per-condition descriptive statistics
5  - Naive two-sample t-test on pairwise similarities
6  - Mann-Whitney U (nonparametric check)
7  - Cohen's d effect size
8  - Output-level bootstrap 95% CI for the difference in mean similarity
9    (corrects for pairwise dependence)
10 """
11 from __future__ import annotations
12
13 import json
14 from itertools import combinations
15 from pathlib import Path
16
17 import numpy as np
18 import pandas as pd
19 import yaml
20 from scipy import stats
21
22
23 def load_config(path: str = "config.yaml") →dict:
24     with open(path, "r") as f:
25         return yaml.safe_load(f)
26
27
28 def cohens_d(a: np.ndarray, b: np.ndarray) →float:
29     """Pooled-SD Cohen's d for two independent samples."""
30     na, nb = len(a), len(b)
31     va, vb = a.var(ddof=1), b.var(ddof=1)
32     pooled_sd = np.sqrt(((na - 1) * va + (nb - 1) * vb) / (na + nb - 2))
33     return (a.mean() - b.mean()) / pooled_sd
34
35
36 def mean_pairwise(embeddings: np.ndarray) →float:
37     """Mean pairwise cosine for an L2-normalized embedding matrix."""
38     n = embeddings.shape[0]
39     sims = [
40         float(embeddings[i] @ embeddings[j])
41         for i, j in combinations(range(n), 2)
42     ]
43     return float(np.mean(sims))
44
45
46 def bootstrap_diff(
47     sparse_emb: np.ndarray,
48     dense_emb: np.ndarray,
49     n_iter: int,
50     rng: np.random.Generator,
51 ) →tuple[float, float, np.ndarray]:
52     """Output-level bootstrap of (mean_sparse - mean_dense).
53
54     Resamples outputs (not pairs) with replacement, recomputes mean
55     pairwise similarity in each condition, returns 95% CI.

```

```

56 """
57 n_s, n_d = sparse_emb.shape[0], dense_emb.shape[0]
58 diffs = np.empty(n_iter, dtype=float)
59 for k in range(n_iter):
60     idx_s = rng.integers(0, n_s, size=n_s)
61     idx_d = rng.integers(0, n_d, size=n_d)
62     ms = mean_pairwise(sparse_emb[idx_s])
63     md = mean_pairwise(dense_emb[idx_d])
64     diffs[k] = ms - md
65 lo, hi = np.percentile(diffs, [2.5, 97.5])
66 return float(lo), float(hi), diffs
67
68
69 def main() →None:
70     cfg = load_config()
71     emb_root = Path(cfg["paths"]["embeddings_dir"])
72     results_root = Path(cfg["paths"]["results_dir"])
73     results_root.mkdir(parents=True, exist_ok=True)
74
75     sparse_emb = np.load(emb_root / "sparse.npy")
76     dense_emb = np.load(emb_root / "dense.npy")
77
78     df = pd.read_csv(results_root / "pairwise.csv")
79     sparse_sims = df.loc[df["condition"] == "sparse", "cosine"].to_numpy()
80     dense_sims = df.loc[df["condition"] == "dense", "cosine"].to_numpy()
81
82     # Descriptive
83     desc = {
84         "sparse": {
85             "n_outputs": int(sparse_emb.shape[0]),
86             "n_pairs": int(len(sparse_sims)),
87             "mean": float(sparse_sims.mean()),
88             "std": float(sparse_sims.std(ddof=1)),
89             "median": float(np.median(sparse_sims)),
90         },
91         "dense": {
92             "n_outputs": int(dense_emb.shape[0]),
93             "n_pairs": int(len(dense_sims)),
94             "mean": float(dense_sims.mean()),
95             "std": float(dense_sims.std(ddof=1)),
96             "median": float(np.median(dense_sims)),
97         },
98     }
99
100     # Naive t-test (note: pairwise dependence means this is optimistic)
101     t_stat, t_p = stats.ttest_ind(sparse_sims, dense_sims, equal_var=False)
102
103     # Nonparametric check
104     u_stat, u_p = stats.mannwhitneyu(
105         sparse_sims, dense_sims, alternative="two-sided"
106     )
107
108     # Effect size
109     d = cohens_d(sparse_sims, dense_sims)
110
111     # Output-level bootstrap (the honest test)
112     rng = np.random.default_rng(cfg["analysis"]["random_seed"])
113     lo, hi, _ = bootstrap_diff(
114         sparse_emb,

```



```
115     dense_emb,
116     n_iter=cfg["analysis"]["bootstrap_iterations"],
117     rng=rng,
118 )
119
120 summary = {
121     "descriptive": desc,
122     "naive_welch_t_test": {"t": float(t_stat), "p": float(t_p)},
123     "mann_whitney_u": {"u": float(u_stat), "p": float(u_p)},
124     "cohens_d": float(d),
125     "bootstrap_diff_in_means": {
126         "point_estimate": float(sparse_sims.mean() - dense_sims.mean()),
127         "ci_low": lo,
128         "ci_high": hi,
129         "n_iter": int(cfg["analysis"]["bootstrap_iterations"]),
130     },
131 }
132
133 with open(results_root / "stats.json", "w") as f:
134     json.dump(summary, f, indent=2)
135
136 # Pretty print
137 print(json.dumps(summary, indent=2))
138
139
140 if __name__ == "__main__":
141     main()
```

10 Plotting

10.1 plot.py

```

1  """Violin + strip plot of pairwise similarity distributions per condition."""
2  from __future__ import annotations
3
4  from pathlib import Path
5
6  import matplotlib.pyplot as plt
7  import pandas as pd
8  import seaborn as sns
9  import yaml
10
11
12  def load_config(path: str = "config.yaml") →dict:
13      with open(path, "r") as f:
14          return yaml.safe_load(f)
15
16
17  def main() →None:
18      cfg = load_config()
19      results_root = Path(cfg["paths"]["results_dir"])
20      df = pd.read_csv(results_root / "pairwise.csv")
21
22      sns.set_theme(style="whitegrid", context="talk")
23      fig, ax = plt.subplots(figsize=(8, 6))
24
25      sns.violinplot(
26          data=df,
27          x="condition",
28          y="cosine",
29          order=["sparse", "dense"],
30          inner="quartile",
31          cut=0,
32          ax=ax,
33      )
34      sns.stripplot(
35          data=df,
36          x="condition",
37          y="cosine",
38          order=["sparse", "dense"],
39          color="black",
40          alpha=0.25,
41          size=2,
42          ax=ax,
43      )
44
45      ax.set_xlabel("Specification condition")
46      ax.set_ylabel("Pairwise cosine similarity")
47      ax.set_title("Output similarity by specification density")
48      fig.tight_layout()
49      fig.savefig(results_root / "plot.png", dpi=200)
50      print(f"Saved {results_root / 'plot.png'}")
51
52
53  if __name__ == "__main__":
54      main()

```

11 Pipeline Orchestrator

11.1 run_all.py

```
1 """Run the full pipeline end-to-end.
2
3 Usage:
4   python run_all.py
5 """
6 from __future__ import annotations
7
8 import subprocess
9 import sys
10
11
12 STEPS = [
13     ("Generating completions via LMStudio", "generate.py"),
14     ("Embedding outputs", "embed.py"),
15     ("Computing pairwise similarities", "similarity.py"),
16     ("Running statistical tests", "stats.py"),
17     ("Plotting", "plot.py"),
18 ]
19
20
21 def main() →None:
22     for title, script in STEPS:
23         print(f"\n=== {title} ({script}) ===")
24         result = subprocess.run([sys.executable, script])
25         if result.returncode != 0:
26             print(f"Step failed: {script}")
27             sys.exit(result.returncode)
28     print("\nPipeline complete. See results/ for outputs.")
29
30
31 if __name__ == "__main__":
32     main()
```

12 README

12.1 README.md

```

1 # Specification Dilemma Experiment
2
3 Small empirical probe for the claim that sparse-specification prompts
4 produce more homogeneous outputs across users than dense-specification
5 prompts.
6
7 See experiment.pdf for the full specification.
8
9 ## Design note: matched pairs
10
11 The prompts in this repo follow a matched-pairs structure. Each of 30
12 imagined users has a fixed underlying intent (audience, thesis, tone,
13 voice, opening move, structural constraint). prompts/dense.json[i]
14 expresses that user's full intent; prompts/sparse.json[i] is what the
15 same user would type when underspecifying -- topic only, in roughly
16 their natural register. The sparse prompts carry no audience, thesis,
17 tone, or structural specification.
18
19 The statistical comparison is unchanged -- cross-user pairwise
20 similarity in each condition -- but the two conditions now sample the
21 same population of underlying intents. This tests the sharper claim:
22 when users with divergent intents underspecify, outputs converge
23 (priors dominate); when they specify fully, outputs diverge (intents
24 dominate).
25
26 ## Setup
27
28 1. Install LMStudio and download a strong instruction-tuned model
29    (e.g. Qwen2.5-72B-Instruct or Llama-3.3-70B-Instruct).
30 2. Start the LMStudio local server (default: localhost:1234).
31 3. Create the environment and install dependencies with uv:
32
33     uv sync
34
35     (or pip install -r requirements.txt inside a venv if not using uv)
36
37 4. Edit config.yaml if your LMStudio model name or port differs from
38    the defaults. If LMStudio is on a remote host, point
39    lmstudio.base_url at that host (e.g. http://<host>:1234/v1).
40
41 5. Smoke-test the endpoint (checks connectivity, seed-honoring, and
42    approximate per-generation latency):
43
44     uv run python smoke_test.py
45
46 ## Running
47
48 Freeze your prompts in prompts/sparse.json and prompts/dense.json
49 before generating anything.
50
51 Then run the full pipeline:
52
53     uv run python run_all.py
54
55 Or run steps individually:

```

```
56
57 uv run python generate.py # LMStudio generations
58 uv run python embed.py # sentence embeddings
59 uv run python similarity.py # pairwise cosine similarities
60 uv run python stats.py # t-test, Mann-Whitney, bootstrap, Cohen's d
61 uv run python plot.py # violin plot
62
63 ## Outputs
64
65 - outputs/{sparse,dense}/NN.txt : raw model completions
66 - embeddings/{sparse,dense}.npy : L2-normalized embedding matrices
67 - results/pairwise.csv : all pairwise similarities
68 - results/stats.json : test statistics and summary
69 - results/plot.png : similarity distribution plot
70
71 ## Interpretation
72
73 A positive result: sparse-condition mean pairwise similarity is
74 meaningfully higher than dense-condition mean similarity, the
75 bootstrap 95% CI on the difference excludes 0, and Cohen's d is
76 large (>0.8).
77
78 A null or inverted result is also interesting and should be reported
79 honestly.
```

13 Essay Integration Notes

The experiment should occupy roughly 300–500 words in the essay itself, including setup, result, and caveat. One paragraph describing the design, one paragraph reporting the result with the plot, and one sentence acknowledging limitations.

The honest framing, to place in a footnote:

This is a single-task, single-model probe with one embedding-based similarity metric. A fuller treatment would need to show the effect holds across tasks, models, and metrics. I ran this to check whether my intuition survived contact with data; report whichever result you find.

If the experiment's word budget starts expanding past 500 words, it has begun competing with the argument rather than serving it. The mechanism section and the “why this is worse than individual decline” section are the essay's center of gravity; the experiment is a short piece of evidentiary scaffolding that lets those sections claim more than pure argument would permit.