ChatGPT

Trishift（GOT-ShiftGen）模型开发与实现深度研究报告（基于 scPRAM / Scouter / Systema / GenePT）

本报告聚焦于 **Trishift 论文所提出的 GOT-ShiftGen（GenePT-Conditioned Optimal-Transport Shift Generator）** 的落地实现：以 **scPRAM** 的“VAE + OT 对齐 + 注意力/个体化位移”思想为潜空间与对齐基础，以 **Scouter** 的“压缩器–生成器（compressor–generator）+ GEARS 风格 autofocus & direction-aware 损失”为表达生成主干，以 **Systema** 的“抵抗 systematic variation 的评估体系（尤其 PearsonΔ 系列）”作为主评估框架，并以 **GenePT** 提供的 LLM 文本嵌入（预下载 pickle）作为**扰动语义条件**。Trishift 论文在摘要中明确描述三阶段训练与双轨评估（传统 GEARS/Scouter 指标 + Systema Pearson-Δ 系列）。

四个仓库的代码基线与可复用资产分析

scPRAM（elan6666/scPRAM）：VAE 潜空间 + OT 配对 + 注意力式位移聚合

任务定位与论文方法要点

scPRAM 的论文强调：用 **VAE** 将高维稀疏表达映射到潜空间，用 **最优传输（OT，Sinkhorn）** 对齐扰动前后“非配对”细胞，再用 **注意力机制** 为待预测细胞计算个体化扰动向量，从而预测扰动响应分布（而不仅是均值）。 [1]

代码结构与关键入口（README）

其 Python API 按 “预处理→训练→预测→评估”组织，核心类为 `scpram.models.SCPRAM`，训练入口 `train_SCPRAM`，预测入口 `predict`。

可直接复用的核心实现点（models.py）

1) **Stage1 可复用：VAE 编码器/解码器骨架**

`SCPRAM` 在 `__init__` 中用 MLP 构建 encoder / decoder，并通过 `encode()` 返回 `(z, mu, logvar)`，这是 Trishift Stage1 的天然实现素材。

2) **OT 配对可复用：POT 的 cost matrix + EMD coupling + argmax 匹配**

scPRAM 的 `predict()` 计算 `M = ot.dist(stim, ctrl)`，再用 `ot.emd()` 得到耦合矩阵 `G`，并通过 `torch.max(G, 0)` 做硬匹配（每个 control 选一个 best matched perturbed），形成 per-cell 位移 `delta_list = stim_new - ctrl`。

> Trishift Stage2/Stage3 需要更一般的 **hard Top-k / kNN matching**（并且通常按“每个 perturbed 找 top-k control”），但该段代码给出了“如何从 OT 耦合矩阵导出硬匹配监督”的最短路径。
>
> 3) **注意力式位移聚合（cosine similarity + top ratio）**
>
> `predict()` 里对 test latent `test_z` 与训练 control latent `ctrl` 做 cosine similarity，选取 top ratio 的控制细胞并归一化权重，再对 `delta_list` 加权求和得到个体化位移再解码。
>
> 这与 Trishift 想要的“个体异质性位移码”在概念上同源：差别在于 Trishift 把“位移的学习”从启发式聚合升级为 **可训练的位移预测器**。

工程依赖与兼容性提示

scPRAM 的依赖中包含 `pot`、`scanpy`、`torch` 等。

但其环境文件偏向较旧 torch（1.13），而 Scouter/Systema 更偏 torch≥2；Trishift 实现建议以 **torch≥2 + Python≥3.10** 为主环境，并对 scPRAM 代码做最小兼容补丁（见后续“性能与工程化”与“文件映射”）。

Scouter（elan6666/scouter）：GenePT 嵌入 + 压缩器–生成器 + GEARS 风格损失

论文要点（方法与训练）
Scouter 论文明确：使用来自 GenePT 的 **1536 维基因文本嵌入**，输入为“随机抽取的 control cell 表达 + 扰动基因 embedding”，通过 compressor–generator 结构输出预测表达；基因 embedding 固定不训练；多基因扰动通过 **embedding 求和**。 [2]
其损失包含 GEARS 风格的 **autofocus**（幂次加权误差）与 **direction-aware**（方向一致性惩罚），并配合训练细节（Adam、指数衰减、early stopping、梯度裁剪等）。 [2]

仓库 API 与关键对象（README）
仓库提供 `ScouterData`（数据与嵌入对齐/切分/DEG 统计）与 `Scouter`（训练、预测、评估）。

可复用的核心模块
1) **ScouterModel：压缩器–生成器的最小实现**
- `self.embd`：固定不可训练的 embedding 参数；
- `encoder(ctrl_exp)`：压缩 control 表达到低维 cell state；
- `generator([cell_state ⊕ gene_emb])`：输出预测表达；
- 多基因扰动：`self.embd[pert_idx].sum(axis=1)`。
这几乎就是 Trishift Stage3 “Scouter-like” 生成器的基底，只需额外拼接 **位移码 Δcz**（Trishift 的关键创新点之一）。

2) **gears_loss：autofocus + direction-aware 的可直接移植实现**
`gears_loss()` 中 autofocus 与 direction-aware 的逐元素形式和组合方式非常清晰，并按 perturbation group 选取 `nonzero_idx_dict[p]` 进行 loss 聚合。
这与 Trishift Stage3 描述（“Scouter/GEARS 风格 autofocus + direction-aware 损失”）一致。

3) **ScouterData：嵌入矩阵与扰动条件的对齐、丢弃未匹配扰动、生成 embedding 索引**
`setup_ad()` 会：
- 取 `adata.obs[key_label]` 中所有 perturbation condition，解析出所有基因 token；
- 在 embedding matrix 行名与 perturbation gene tokens 间取交集；
- 若有未覆盖基因，则删除相关 perturbation 条件；
- 在 `adata.obs` 新增 `key_embd_index` 列：每个 cell 的 perturbation 对应 embedding 行索引列表。
这可作为 Trishift 的“GenePT 嵌入对齐与样本清洗”主实现来源。

4) **训练循环（Scouter.train）：工程细节可复用**
- `BalancedDataset` 生成 (ctrl_expr, true_expr, pert_idx, bcode)；
- 训练采用 Adam、ExponentialLR、early stopping、梯度裁剪；
Trishift Stage3 可在该训练框架上最小改动：将 dataset 从“随机 ctrl”替换为“OT 匹配 ctrl”。

许可（MIT）
Scouter 仓库使用 MIT License。

Systema（elan6666/systema）：评估框架与 PearsonΔ（自定义 reference）指标体系

论文要点（为何必须用 Systema）
Systema 指出：很多 perturbation response prediction 的高分可能主要来自 **systematic variation**（perturbed vs control 的系统性差异，如 panel selection bias、confounders 等），导致传统 reference-based 指标对真实“扰动特异性效应”评估不足；因此 Systema 建议采用更能突出扰动特异性的 reference（例如 **perturbed centroid reference**）并提出相应评估框架。 [3]

仓库定位（README）
Systema 仓库实现了统一 pipeline、若干方法与 baseline，并强调 evaluation 文件夹提供易用评估实现，包含：
1) **Pearson correlation on delta profiles using centroid of perturbed cells as reference**；
2) **centroid accuracy**（预测是否更接近其正确 perturbation centroid）。

评估实现可复用点
1) evaluation/README.md 给出了 PearsonΔ（自定义 reference）接口形态：`pearson_delta_reference_metrics(X_true, X_pred, reference, top20_de_idxs)`。
2) `centroid_accuracy.py` 用欧氏距离矩阵计算每个方法在每个 perturbation 上“相对其他 centroids 的正确性比例”，是 Trishift 双轨评估中非常合适的补充（不仅看相关，还看 landscape）。

需要注意的“微补丁点”
本 fork 中 `pearson_delta_reference_metrics.py` 当前实现缺少 `top20_de_idxs` 入参（与 evaluation/README 不一致），且内部引用了未定义变量 `top20_de_idxs`。
因此 Trishift 集成 Systema 时，应当以 README 期望的函数签名为准，在 Trishift 仓库内做一次“兼容包装/修复实现”（见后续“文件映射表”与“评估实现建议”）。

GenePT（elan6666/GenePT）：Gene embedding 数据源与 pickle 格式线索

论文与数据发布要点
GenePT 提出以 NCBI gene summary 文本，调用 GPT embedding API 生成每个基因的向量表示，并进一步构造 cell embedding；其核心优势是无需大规模表达预训练就能获得可用的语义嵌入。 4
GenePT 官方在 Zenodo 发布了预计算 embeddings（含更新版本 GenePT_emebdding_v2.zip），并标注为数据集 DOI。 5

仓库代码提示：pickle 的“字典→numpy array”形态
GenePT 的 `gene_embeddings_examples.ipynb` 展示了 `GPT_DIM = 1536`，并将 `gpt_gene_name_to_embedding_clean_text` 以 pickle 形式保存（字典：gene_name → embedding numpy array）。
这与用户给定的本地路径 `GenePT_gene_embedding_ada_text.pickle` 的命名高度一致（很可能就是 GenePT 1536 维 ada-002 embedding 的字典或 DataFrame）。

Trishift 如何借鉴四个工作：概念/方法/代码层面的可行集成逻辑

Trishift（GOT-ShiftGen）在摘要中把自身定位为“三阶段训练 + 双轨评估”的工程组合：Stage1 用 VAE 学潜空间、Stage2 在潜空间做 OT + hard kNN 得到 ΔzOT 监督并训练 GenePT 条件位移预测器、Stage3 用新的 compressor–generator 在 OT 匹配对上训练表达生成，并同时报告 GEARS/Scouter 与 Systema 指标。

下述对应关系可视为“从四个仓库中抽取最小可复用闭环”的设计动机与可行性论证：

从 scPRAM 到 Trishift：把“OT+注意力求位移”升级为“OT+可学习位移预测器”

- scPRAM 的关键技术栈是 **VAE→OT 配对→注意力加权位移**。 1
- 其代码已经给出了 VAE 与 OT coupling 的具体实现路径，并且在 `predict()` 内把 OT 匹配得到的 per-cell shift（delta_list）作为后续注意力聚合的“value”。
- Trishift Stage2 的核心变化是：不再把位移仅当作“启发式 attention 输出”，而是把 OT 导出的位移当作 **监督信号 ΔzOT**，训练一个 **以 GenePT embedding 为条件的位移预测器 fψ([z_ctrl ⊕ e(p)]) → Δz**。这可以显著提升对“未见扰动基因”的泛化能力，因为 shift predictor 的输入包含了可泛化的扰动语义 e(p)。

- 工程可行性：scPRAM 已能产出稳定 latent 与 OT 匹配；在此基础上加一层 MLP（shift predictor）属于“低耦合增量”，并不会破坏 scPRAM 的主要数据流。

从 GenePT 到 Trishift：用“扰动语义向量”解决 unseen perturbation 的可表示性

- scPRAM 的 attention 主要解决“不同 cell state 该如何选取匹配的 perturbation direction”，但它并不直接提供“未见基因扰动”的表示学习机制。 [1]
- Scouter 论文明确指出：类别变量“基因身份”难以泛化，而 **LLM gene embedding** 将文本语义映射到连续向量空间，使 unseen 基因可被放在与 seen 基因同一漂移几何空间中；并且 Scouter 用的 GenePT embedding 长度为 1536。 [6]
- Trishift 把 GenePT embedding 放进 Stage2/Stage3，作为扰动条件 e(p)，其合理性在于：
  1) Stage2 学的是“该 perturbation 对 latent space 的位移模式”；
  2) Stage3 学的是“在 expression space 上的生成映射”；
  两者都需要一个可泛化的扰动条件编码。

从 Scouter 到 Trishift：把 generator 学习从“随机 ctrl triplet”改为“OT 匹配对/位移码增强”

- Scouter 训练的基本单位是 triplet（perturbed cell x_i, gene embedding E(p_i), 随机 ctrl cell c_i），通过 compressor→generator 预测表达。 [2]
- Scouter 的实现中，`ScouterModel.forward()` 正是执行 `S_i = encoder(c_i)`、拼接 `E(p_i)`、再 generator 输出。
- Trishift Stage3 在此基础上做两点关键增强：
  1) **把“随机 ctrl”替换为“OT 诱导匹配 ctrl”**，使训练样本对更贴近真实“同一细胞状态的扰动前后”。
  2) **增加“位移码 Δcz”作为显式输入**（由 Stage2 shift predictor 输出），让 generator 同时看到“cell state（由 compressor 得到）+ perturbation semantics（GenePT）+ individualized shift（latent-level）”。
- 这一改造在代码上非常可控：只需在 Scouter 的 `ScouterModel` 拼接向量时额外 concat 一段 shift code，并用新 Dataset 提供匹配 ctrl。scouter/_model.py 的 MLP 构建器 `_build_mlp` 可直接复用。

从 Systema 到 Trishift：用 perturbed centroid reference 的 PearsonΔ 抵抗“虚高的泛化分数”

- Systema 论文与仓库都强调：传统 reference-based 指标可能对 systematic variation 敏感，导致高估模型对 perturbation-specific effects 的刻画能力；Systema 提供 perturbed centroid reference 并据此计算 PearsonΔ/Δ20。 [3]
- Trishift 的双轨评估设计（传统 nMSE/1-nPCC + Systema Pearson-Δpert）正是为了避免仅在 reference-sensitive 指标上“被 systematic variation 虚高”。

Trishift 实现的推荐仓库结构与最小重写集成策略

本节给出一个**面向实现落地**的 Trishift 仓库结构（假设新仓库名为 `trishift`），核心目标是：

- **最大化复用**：
- scouter：作为 Stage3 基座（模型结构 + loss + 数据处理）；
- scPRAM：作为 Stage1 VAE 与 OT→硬匹配监督的最短实现来源；
- systema：主要复用 evaluation（PearsonΔ、centroid accuracy）；
- GenePT：只作为 embedding 数据与格式参考（不在 Trishift 内重新生成 embedding）。 [7]

建议目录树（含 third_party 子模块）

```
trishift/
  README.md
  AGENTS.md
  pyproject.toml
  requirements/
    base.txt
    dev.txt
  configs/
    default.yaml
    data/
      norman2019.yaml
  third_party/                          # git submodules（只读，尽量不改）
    scouter/
    scPRAM/
    systema/
    GenePT/
  src/trishift/
    __init__.py
    data/
      adata_preprocess.py               # HVG/normalize/log1p；与 scouter/scPRAM 的输入对齐
      splits.py                         # by-perturbation split（复用 ScouterData.split_* 逻
辑）
      genept_embedding_loader.py        # 读取本地 pickle → DataFrame/Tensor
      perturbation_encoding.py          # e(p)=Norm(Pool(E(g)))
    stage1/
      vae_adapter.py                    # 轻封装 scPRAM 的 SCPRAM：输出 mu 作为确定性 z
      latent_cache.py                   # 预计算/缓存 z_ctrl, z_pert（避免反复编码）
    stage2/
      ot_sinkhorn.py                    # POT Sinkhorn + top-k matching（从 scPRAM 的 OT 代码改
造）
      shift_dataset.py                  # (ctrl_idx, pert_idx, pert_cond_id) + delta_z on-the-
fly
      shift_predictor.py                # fψ([z_ctrl ⊕ e(p)]) -> Δz
      train_shift.py
    stage3/
      trishift_generator.py             # 改造 scouter/_model.py：concat 额外 shift code
      matched_dataset.py                # 用 OT 匹配对替代 BalancedDataset
      loss.py                           # 复用 scouter/_utils.gears_loss（必要时薄封装）
      train_generator.py
    evaluation/
      metrics_gears_scouter.py          # 复用 Scouter 的 NormMSE/Pearson/Spearman 思路
      metrics_systema.py                # 复用/修复 Systema PearsonΔ + centroid accuracy
      run_eval.py
    utils/
      seed.py
      device.py
      logging.py
  scripts/
    run_stage1.py
```

```
run_stage2.py
run_stage3.py
run_eval.py
```

集成方式首选：submodule + Adapter（而非复制粘贴）

- **third_party 只读子模块**：保留原始历史与许可证信息，便于对照论文与上游实现逻辑；
- **Adapter 层**：在 `src/trishift/` 下用极薄封装适配接口（例如把 scPRAM 的 `get_latent_adata()` 改成返回 `mu`），尽量不直接“侵入式修改 third_party”。
- 只有当上游实现存在明显接口不一致（例如 Systema 的 PearsonΔ 实现缺失入参）时，才在 Trishift 内写“修复版 wrapper”，并在 mapping 表中明确来源与改动点。

Trishift 文件级复用映射表（Trishift → 来源仓库文件）

下表是“建议实现方案”的**显式映射**：即 Trishift 仓库中哪些模块应直接借用/改造自哪些源文件。为满足“最大复用、最小重写”，表中区分三类复用方式：

- **直接复用**：作为 submodule 原样引入（不改）；
- **轻改造（Adapter）**：复制或继承后做小改动（接口/维度/拼接）；
- **新实现**：源仓库没有对应功能，但会引用其数据结构/函数。

Trishift 目标文件/模块	复用方式	来源仓库 → 文件	复用/改造要点
`third_party/scPRAM/scpram/models.py`	直接复用（只读）	scPRAM → `scpram/models.py`	作为 Stage1 VAE 的参考实现（En
`src/trishift/stage1/vae_adapter.py`	轻改造（Adapter）	scPRAM → `scpram/models.py`	复用 `SCPRAM.encode()` 输出 `(z` `z=mu`（论文要求）。
`src/trishift/stage2/ot_sinkhorn.py`	轻改造	scPRAM → `scpram/models.py`（OT+硬匹配段）	将 `ot.emd` 升级为 Sinkhorn（符 pert”改为“每个 pert 取 top-k c
`src/trishift/stage2/shift_predictor.py`	新实现（小）	参考 scPRAM 的位移定义 + Trishift 论文	实现 MLP fψ：输入 `[z_ctrl ⊕` `z_ctrl`。
`third_party/scouter/scouter/_model.py`	直接复用（只读）	Scouter → `scouter/_model.py`	作为 Stage3 generator 的基座：e generator。
`src/trishift/stage3/trishift_generator.py`	轻改造	Scouter → `scouter/_model.py`	在 `forward()` 中把输入从 `[cel` `gene_emb ⊕ shift_code]`，并相
`third_party/scouter/scouter/_utils.py::gears_loss`	直接复用（只读）	Scouter → `scouter/_utils.py`	Stage3 使用同款 autofocus + dir dropout genes 聚合 loss” 的策略
`src/trishift/stage3/matched_dataset.py`	轻改造	Scouter → `_datasets.py::BalancedDataset`	将“随机 ctrl pairing”改为“OT 致，减少训练循环改动面。
`third_party/scouter/scouter/ScouterData.py`	直接复用（只读）	Scouter → `ScouterData.py`	负责：embedding/扰动条件匹配 train/val/test、DEG/非零基因索
`src/trishift/data/genept_embedding_loader.py`	轻改造	GenePT → `gene_embeddings_examples.ipynb`（格式线索）	兼容“pickle dict→embedding” 行（0 向量）；转换为 float32 ten

Trishift 目标文件/模块	复用方式	来源仓库 → 文件	复用/改造要点
`third_party/systema/evaluation/centroid_accuracy.py`	直接复用（只读）	Systema → `evaluation/centroid_accuracy.py`	Trishift 双轨评估之一：centroid
`src/trishift/evaluation/metrics_systema.py`	轻改造（修复包装）	Systema → `evaluation/README.md` + `pearson_delta_reference_metrics.py`	以 README 定义为准实现 `pearson_delta_reference_metri` 避免源文件入参缺失问题。

说明：Systema 仓库本 fork 未发现 LICENSE 文件（至少根目录无 `LICENSE` / `LICENSE.md`），建议以“submodule + 最小包装”的方式复用，并在正式发布 Trishift 仓库前核对上游许可或与作者确认。

GenePT 预下载嵌入（pickle）在 Trishift 中的加载、预处理与高效集成

用户提供的嵌入文件路径（Windows）：

- `E:\CODE\GenePert-main\GenePert-main\data\GenePT_emebdding_v2\GenePT_gene_embedding_ada_text.pickle`
- `E:\CODE\GenePert-main\GenePert-main\data\GenePT_emebdding_v2\GenePT_gene_protein_embedding_model_3_text.pickle`

结合 GenePT 的公开数据描述，GenePT embedding 数据集以 zip 发布并与 GenePT 预印本 DOI 关联。[7]
GenePT notebook 显示常见格式为 `dict[str, np.ndarray]` 且 `GPT_DIM=1536`。
Scouter 侧要求 embedding matrix 的行名包含所有 perturbation gene token，并且必须能找到 `'ctrl'`（作为对照占位）。

推荐加载与标准化策略

下面给出一个“健壮型 loader”思路（支持 dict 或 DataFrame；自动补 `ctrl`；可选 L2 归一化；可缓存为 `.pt` 以加速后续启动）。示例代码仅为实现指导：

```
# src/trishift/data/genept_embedding_loader.py
from __future__ import annotations
import pickle
import numpy as np
import pandas as pd
import torch
from pathlib import Path

def load_genept_pickle(path: str | Path, add_ctrl: bool = True, l2_normalize: bool = True) ->
pd.DataFrame:
    path = Path(path)
    with path.open("rb") as f:
        obj = pickle.load(f)

    # 1) 兼容：pickle 里可能是 dict 或 DataFrame
    if isinstance(obj, pd.DataFrame):
        embd_df = obj.copy()
    elif isinstance(obj, dict):
```

```
        # dict: gene -> vector
        embd_df = pd.DataFrame.from_dict(obj, orient="index")
    else:
        raise TypeError(f"Unsupported pickle type: {type(obj)}")

    # 2) dtype & NaN 处理
    embd_df = embd_df.apply(pd.to_numeric, errors="coerce").fillna(0.0)
    embd_df = embd_df.astype(np.float32)

    # 3) 补齐 ctrl token（Scouter/Trishift 需要）
    if add_ctrl and "ctrl" not in embd_df.index:
        d = embd_df.shape[1]
        embd_df.loc["ctrl"] = np.zeros(d, dtype=np.float32)

    # 4) 可选：每行 L2 normalize（使 e(p) 尺度稳定）
    if l2_normalize:
        x = embd_df.values
        norm = np.linalg.norm(x, axis=1, keepdims=True) + 1e-12
        embd_df.iloc[:, :] = x / norm

    return embd_df

def to_torch_embedding(embd_df: pd.DataFrame, device: str = "cpu") -> tuple[torch.Tensor,
dict[str, int]]:
    gene2idx = {g: i for i, g in enumerate(embd_df.index)}
    embd = torch.tensor(embd_df.values, dtype=torch.float32, device=device)
    return embd, gene2idx
```

在 Trishift pipeline 中的注入点

注入点 A：ScouterData.setup_ad 用于“扰动条件→embedding 索引”
将 `embd_df` 作为 `ScouterData(adata, embd_df, key_label, key_var_genename)` 的 `embd` 输入，然后调用 `setup_ad(key_embd_index='embd_index')`，让 `adata.obs['embd_index']` 变成“该细胞 perturbation 的 embedding 行索引列表”。

注入点 B：Trishift 的 perturbation encoding：e(p)=Norm(Pool(E(g)))
- Scouter 的做法等价于 `Pool=sum` 且不显式 Norm（forward 中直接 sum）。
- Trishift 摘要中强调 `Pool` 与 `Norm`（并将 e(p) 用于 Stage2/Stage3）：
建议实现为：
1) 单基因：`e(p)=E(g)`（如已 L2 normalize 则无需再做）；
2) 多基因：`e(p)=L2Norm(Σ E(g_i))` 或 `L2Norm(mean(E(g_i)))`。
工程上可把 `e(p)` 预计算为 `cond2embd_vec`（每个 perturbation condition 一个向量），减少训练期重复 gather+sum。

注入点 C：Stage2 shift predictor 的条件输入
Stage2 的 fψ 需要 `[z_ctrl ⊕ e(p)]`。如果 `e(p)` 预计算好并保存为 `cond_id` 索引，则 dataset 只需返回 `cond_id`，训练时从 embedding table gather 即可（节省 IO）。

两个 pickle 的选择与一致性校验

`ada_text` 通常对应 1536 维（与 GenePT notebook 的 GPT_DIM 一致）。
`gene_protein_embedding_model_3_text` 可能对应“更大/更新的 embedding 模型 + 蛋白信息增强”（GenePT README 提到 2024-03 上传了含蛋白信息与更新 embedding 模型的新版本）。
因此建议在 loader 后做两类一致性检查：

1) **维度检查**：记录 `d_e = embd_df.shape[1]`，并写入 config；Stage2/Stage3 网络输入维度据此自动推断。
2) **覆盖率检查**：统计 `unmatched_genes`（ScouterData 会提示并删除条件）。
3) **ctrl token 检查**：确保 embedding matrix 含 `'ctrl'`，否则 ScouterData 会报错。

训练、评估与性能优化建议（效率 / 内存 / 可扩展性）

训练流程（与 Trishift three-stage 对齐）

以下为与论文摘要一致的工程化流程（每步都可映射到复用模块）。

Stage1：VAE 学习可对齐潜空间（control & perturbed 共享）
- 直接复用 scPRAM 的 `SCPRAM`（encoder/decoder + KL/rec）。
- Trishift 需要确定性潜变量 `z(x)=mu`，而 scPRAM 的 `get_latent_adata()` 目前用的是采样 `z`（`encode(x)[0]`）。
建议在 `vae_adapter` 中改为取 `encode(x)[1]`（mu）并缓存。

Stage2：潜空间 OT + hard Top-k，构造 ΔzOT 监督并训练 shift predictor
- 复用 scPRAM 的 OT coding pattern（`ot.dist`→coupling→硬匹配），但将 `emd` 改为 **Sinkhorn**（Trishift 与 scPRAM 论文都强调 Sinkhorn OT）。[1]
- 用 hard Top-k 生成多对匹配，实现“监督密度/数据增广”。
- shift predictor 是小 MLP，可参考 ScouterModel 的 MLP 构建方式（BatchNorm + SELU + AlphaDropout 可选）。

Stage3：新 compressor–generator（Scouter-like），输入加入 shift code
- 改造 ScouterModel：把输入拼接扩展到三段：`[cell_state ⊕ e(p) ⊕ Δcz]`。
- 损失直接复用 `gears_loss`（autofocus + direction-aware + nonzero gene selection）。
- dataset 改造：用 OT 匹配 ctrl 替代随机 ctrl（原 Scouter 用 `BalancedDataset` 做随机 ctrl）。

OT 与大数据规模下的性能瓶颈与优化

OT 是 Trishift 相比 Scouter 的主要新增“潜在重计算点”。scPRAM 的实现显示，直接构造完整 cost matrix 并求 EMD coupling 是可行但可能偏重（`numItermax` 也较大）。

建议的工程优化（按收益从高到低）：

1) **潜变量缓存（强烈建议）**
Stage2/Stage3 都需要 `z(ctrl)`：预先对所有 cells 计算 `mu` 并写入 `adata.obsm['z_mu']` 或单独 `.npy/.pt`，避免在 OT 与训练循环里反复跑 encoder。
2) **按 perturbation 分组的子采样策略**
对每个 perturbation p：从全体 ctrl 采样 `n_ctrl_sample`（如 512–4096），从 P(p) 采样 `n_pert_sample`；在样本规模受控的子问题上求 Sinkhorn coupling，再做 top-k。这样 OT 内存从 O(N_ctrl · N_pert) 降到 O(n_ctrl_sample · n_pert_sample)。
3) **“候选集缩小→局部 OT”**

先用 kNN（在 latent space）为每个 perturbed cell 找候选 ctrl 集合（例如 256 个），再只在候选集上做稀疏/局部 OT 或直接 top-k（近似）。这会弱化“全局质量守恒”的 OT 语义，但在大规模场景往往是必要折衷。
4) **数值与设备优化**
- cost matrix 用 float32；coupling/权重可用 float16 存储；
- 若用 GPU Sinkhorn（例如用支持 GPU 的实现/库），可以把 OT 变成吞吐瓶颈而非内存瓶颈。
5) **top-k 的实现避免保存完整 coupling**
当 coupling 过大时，不应保存完整 Π；而应流式计算每列（或每个 perturbed cell） top-k。若使用常规 POT `sinkhorn` 返回 dense matrix，则需在采样层面确保可承载。

表达矩阵（X）导致的内存压力与优化

Scouter 的 `BalancedDataset` 在初始化时把 `adata.X.toarray()` 整体转成 dense，这在细胞数很大时会带来数 GB 级内存风险。

Trishift 建议：

- 若 `adata.X` 是 CSR sparse：在 `__getitem__` 中对单行 `.A1` 或 `.toarray()`，而不是全量 densify；
- 或将数据预处理成 float32 dense memmap（按行分块），用 `numpy.memmap` + PyTorch Dataset 做按需读取；
- 训练中启用：`pin_memory=True`、`num_workers>0`、混合精度（AMP）与梯度累积（在 batch size 受限时）。

评估实现建议：必须同时报告“传统指标 + Systema 指标”

传统指标（Scouter/GEARS 风格）
Scouter 论文与实现都采用 normalized MSE 与（1-）normalized PCC on top DEGs 等指标，并且给出 sampling-based prediction（K=300 control cells）稳定策略。 [2]
代码侧 `Scouter.evaluate()` 给出 per-condition 的 NormMSE、Pearson（基于 true-ctrl 与 pred-ctrl 的相关）、Spearman 等。

Systema 指标（PearsonΔ with perturbed centroid reference + centroid accuracy）
Systema 论文强调 perturbed centroid reference 能更突出扰动特异性，并显示用此 reference 后分数显著降低，从而揭示“真正泛化更难”。 [3]
实现侧建议至少纳入：
- `pearson_delta_reference_metrics()`（修复版）
- `calculate_centroid_accuracies()`（centroid accuracy）

使用 Codex 等 AI 编码助手构建 Trishift 的分步指南（最小手写代码策略）

本节给的是“把 Trishift 组装为可运行仓库”的一种高效率工作流，核心原则：**让 AI 助手做机械性搬运、重构与补全；人只做接口决策、数据格式确认与实验验证**。

Codex 能力边界与工作形态（用于制定协作方式）

官方描述中，Codex 可以读取并修改代码、运行命令与测试，并可通过 CLI/IDE 或云端任务形式工作。 [8]
OpenAI 建议在仓库中提供类似 `AGENTS.md` 的指引文件，帮助 Codex 按项目习惯运行测试、遵循规范。 [9]

实操步骤（建议按 PR/里程碑拆分）

里程碑 A：把 third_party 引进来 + 跑通 Scouter baseline（零改动）
1) 创建 Trishift 空仓库，添加 submodules：`third_party/scouter`、`third_party/scPRAM`、`third_party/systema`、`third_party/GenePT`。
2) 让 Codex 生成 `pyproject.toml` 与 `requirements/base.txt`，以 torch≥2、python≥3.10 为主（与 Scouter setup.py、Systema requirements 对齐）。
3) 让 Codex 写一个 `scripts/run_scouter_baseline.py`：调用 `third_party/scouter` 的 API（`ScouterData.setup_ad()`、`split_Train_Val_Test()`、`gene_ranks()`、`get_dropout_non_zero_genes()`、`Scouter.train()`）。
4) 用合成小数据或 Scouter demo 数据先跑通（Scouter 提供 `adamson_small()` / `embedding_small()`）。

推荐给 Codex 的提示词（示例）
- “在 `scripts/run_scouter_baseline.py` 里实现一个最小可运行脚本：加载 demo 数据、训练 1 epoch、跑 evaluate() 并打印结果。只使用 third_party/scouter 的公开 API，不要复制代码。”

里程碑 B：Stage1（VAE）落地：复用 scPRAM 训练并缓存 `mu`
1) 让 Codex 在 `src/trishift/stage1/vae_adapter.py` 中封装 `SCPRAM`：
- 提供 `fit(adata)`：训练 VAE（直接复用 `train_SCPRAM`）；
- 提供 `encode_mu(X)`：返回 `mu`；
- 提供 `decode(z)`：复用 decoder。
2) 让 Codex 写 `latent_cache.py`：遍历 `adata` 分批编码 `mu`，写入 `adata.obsm['z_mu']` 或 `.pt` 文件。

关键校验点（人工确认）
- scPRAM 原 `get_latent_adata()` 用的是采样 `z`；Trishift 需要确定性 `mu`。

里程碑 C：Stage2（OT + shift predictor）
1) 让 Codex 在 `ot_sinkhorn.py` 写一个函数：
- 输入：`Z_ctrl (n0,dz)`，`Z_pert (n1,dz)`
- 输出：对每个 pert cell 的 top-k control indices（以及可选权重）。
参考 scPRAM 的 `ot.dist` 与 coupling→argmax 结构，但改成 sinkhorn + top-k。 [1]
2) 让 Codex 写 `shift_predictor.py`（MLP），并写 `train_shift.py`：
- dataset 产出 `(ctrl_idx, pert_idx, cond_id)`
- on-the-fly 计算 `ΔzOT = z_pert - z_ctrl`
- loss 用 MSE（与 Trishift 描述一致）。
3) 让 Codex 写 `genept_embedding_loader.py`（读取用户本地 pickle、补 ctrl、归一化、写缓存）。

里程碑 D：Stage3（Trishift generator）
1) 让 Codex 基于 scouter/_model.py 生成 `trishift_generator.py`：
- 新 forward：`shift = fψ([z_ctrl ⊕ e(p)])`（或从 dataset 直接给 shift）
- `cell_state = encoder(ctrl_expr)`
- `pred = generator([cell_state ⊕ e(p) ⊕ shift])`。
2) loss：直接调用 scouter/_utils.py 的 `gears_loss`（必要时薄封装以适配 group/idx_dict）。
3) dataset：由 `BalancedDataset` 改造为 `MatchedDataset`（输入为 OT 匹配对），保持返回 tuple 结构最大相似，复用 Scouter.train 的大部分逻辑。

里程碑 E：Systema 评估对齐
1) 让 Codex 在 `metrics_systema.py` 实现修复版 PearsonΔ：按 Systema evaluation/README 的函数签名实现，并写单测。
2) 直接复用 `centroid_accuracy.py`，把 Trishift 的预测整理为 Systema 期待的 MultiIndex DataFrame 格式。

AGENTS.md（给 Codex 的“仓库导航与测试指南”模板）

结合 OpenAI 文档对 Codex 工作方式的描述，建议在仓库根目录写 `AGENTS.md`，内容包括：如何安装依赖、如何拿 demo 数据跑通、如何跑 unit tests、如何跑小规模 smoke test。[10]

四仓库能力对照表（功能模块与评估指标）

功能/代码模块对照

仓库	主要任务	关键模块（代码层）	Trishift 中的角色
scPRAM	跨 cell type / species 的扰动响应预测（强调分布/异质性）[1]	`SCPRAM` VAE；OT coupling→硬匹配；cosine attention 聚合位移	Stage1 潜空间学习；Stage2 “OT→硬匹配监督”的实现母板
Scouter	未见基因扰动的表达预测（LLM gene embedding）[2]	`ScouterModel` compressor–generator；`gears_loss`；`ScouterData.setup_ad`	Stage3 生成器主干与损失函数；数据对齐/DEG/非零基因索引生成
Systema	评估框架（抵抗 systematic variation）[3]	`pearson_delta_reference_metrics`（需修复包装）；`calculate_centroid_accuracies`	Trishift 双轨评估核心（Pearson-Δpert + centroid accuracy）
GenePT	gene embedding 数据生成与发布[11]	pickle dict 保存 (gene→embedding)，常见 1536 维	提供扰动语义向量：Stage2 & Stage3 的条件输入

评估指标对照（建议 Trishift 输出）

指标族	来源/依据	目的	Trishift 输出建议
nMSE、(1-)nPCC（Top DEGs）	Scouter/GEARS 传统评估规范[2]	与主流基线可比、衡量 top DEG 预测质量	必报：按 perturbation 汇总 mean/median + CI
PearsonΔ / PearsonΔ20（custom reference）	Systema evaluation（perturbed centroid reference）[3]	降低 systematic variation 导致的“虚高”相关性	必报：以 perturbed centroid reference 计算 Pearson-Δpert、Pearson-Δ20pert
Centroid accuracy	Systema evaluation	看是否能恢复 perturbation landscape、区分不同 perturbation	强烈建议：每方法每 perturbation 的 accuracy + overall mean

Trishift 架构与数据流（文本流程图）

模型架构（Stage1–3）

```
          +------------------------------+
          |  Input: control expr x_pre   |
          |         perturbation p       |
          +---------------+--------------+
                          |
                          v
   Stage1 (VAE, from scPRAM)  :  z = mu_phi(x)    (cache)
     - encoder: x -> (mu, logvar)
     - decoder: z -> x_hat  (Stage3 不一定用)
                          |
        +-----------------+-----------------+
        |                                   |
        v                                   v
Stage2 (OT + Shift Predictor)        GenePT embedding
  - per perturbation p:                - load pickle dict/DataFrame
    OT between Z_ctrl and Z_p          - add 'ctrl' row, normalize
    hard top-k matching -> pairs       - build e(p)=Norm(Pool(E(g)))
  - supervision: Δz_OT = z_p - z_ctrl
  - train f_psi([z_ctrl ⊕ e(p)]) -> Δz_pred
        |
        v
Stage3 (Scouter-like Generator, modified)
  - compressor C_ω: x_pre -> s
  - concat: [s ⊕ e(p) ⊕ Δz_pred]
  - generator D_η -> x_post_hat
  - loss: autofocus + direction-aware (from Scouter/GEARS)
```

评估数据流（双轨）

```
Predictions (x_hat_post per perturbation)
   |
   +--> Traditional (Scouter/GEARS-style):
   |      nMSE, (1-)nPCC on top DEGs
   |
   +--> Systema-style:
          reference = perturbed centroid (avg of per-pert centroids)
          PearsonΔ / PearsonΔ20
          centroid accuracy
```

（上述流程与 Trishift 摘要所述“三阶段 + 双轨评估”一致。）

---

1 https://academic.oup.com/bioinformatics/article/40/5/btae265/7646141
https://academic.oup.com/bioinformatics/article/40/5/btae265/7646141

2 6 https://www.nature.com/articles/s43588-025-00912-8
https://www.nature.com/articles/s43588-025-00912-8

3 https://www.nature.com/articles/s41587-025-02777-8
https://www.nature.com/articles/s41587-025-02777-8

4 11 https://pmc.ncbi.nlm.nih.gov/articles/PMC10614824/
https://pmc.ncbi.nlm.nih.gov/articles/PMC10614824/

5 7 https://zenodo.org/records/10833191
https://zenodo.org/records/10833191

8 https://openai.com/codex/
https://openai.com/codex/

9 10 https://openai.com/index/introducing-codex/
https://openai.com/index/introducing-codex/