

大作业报告 -- Diffusers [1] 重构与扩展

袁晨圃

一、缺陷诊断

1.1 利用大模型发现问题

与 Deep-Wiki [2] 多轮对话，整理成重构文档

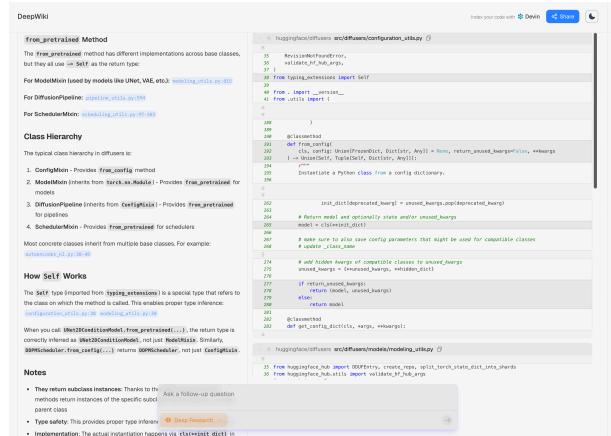


图 1 对话截图

文档节选：

```
## 现有设计模式识别

### 1. 注册表模式 (Registry Pattern)
`_AttentionBackendRegistry` 已经实现了注册表模式 diffusers:206-248：

```python
class _AttentionBackendRegistry:
 _backends = {}
 _constraints = {}

 @classmethod
 def register(cls, backend, constraints=None, supports_context_parallel=False):
 ...
```

### 2. 策略模式 (Strategy Pattern)
不同的attention后端实现（FlashAttention、xFormers、Native等）是不同的算法策略 diffusers:1323-1369。

### 3. 分发器模式 (Dispatcher Pattern)
`dispatch_attention_fn()` 作为分发器，根据backend选择具体实现 diffusers:305-351。

## 重构计划

### 1. 抽象策略接口

```python
from abc import ABC, abstractmethod

class AttentionStrategy(ABC):
 @abstractmethod
 def compute_attention(self, query, key, value, **kwargs): ...
 @abstractmethod
 def validate_constraints(self, **kwargs): ...

class FlashAttentionStrategy(AttentionStrategy):
 def compute_attention(self, query, key, value, **kwargs):
 ...

 def validate_constraints(self, **kwargs):
 ...
```

```

2. 策略工厂

```
```python
class AttentionStrategyFactory:
 _strategies = {}

 @classmethod
 def register_strategy(cls, backend_name: str, strategy_class: type):
 cls._strategies[backend_name] = strategy_class

 @classmethod
 def create_strategy(cls, backend_name: str) -> AttentionStrategy:
 if backend_name not in cls._strategies:
 raise ValueError(f"Unknown backend: {backend_name}")
 return cls._strategies[backend_name]()

注册策略
AttentionStrategyFactory.register_strategy("flash", FlashAttentionStrategy)
AttentionStrategyFactory.register_strategy("xformers", XFormersAttentionStrategy)
```

```

二、重构采用的设计模式介绍

2.1 建造者模式 (Builder Pattern)

① 建造者模式

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

2.1.1 引入

假设有这样一个复杂对象，在对其进行构造时需要构造众多成员变量和嵌套对象。这些初始化代码通常深藏于一个包含众多参数的构造函数中，且散落在客户端代码的多个位置。

建造者模式的解决方案

将对象构造代码从产品类中抽取出来，并将其放在一个名为建造者的独立对象中。

将对象构造过程划分为一组步骤，比如 `buildWalls` 创建墙壁和 `buildDoor` 创建房门等。每次创建对象时，都需要通过建造者对象执行一系列步骤。重点在于无需调用所有步骤，而只需调用创建特定对象配置所需的那些步骤即可。

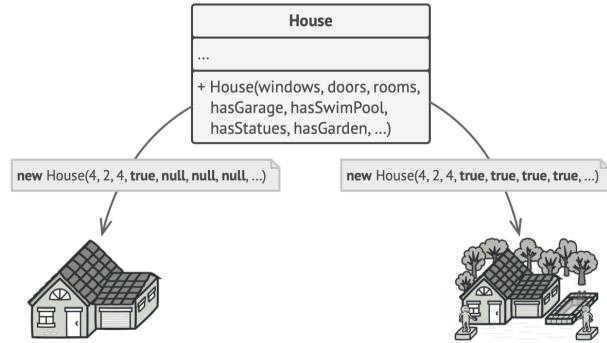


图 2 一个有复杂构造函数的 House 类

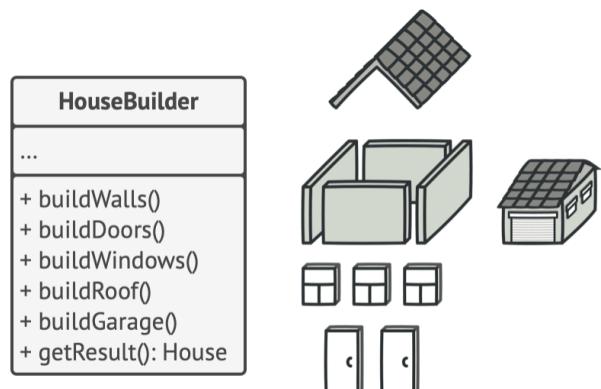


图 3 建造者：HouseBuilder

2.1.2 在代码中的应用

统一 DiffusionPipeline 各组件的构建过程，解决训练脚本中的代码重复和不一致问题。

核心类：DiffusionPipelineBuilder，提供链式配置和组件管理

DiffusionPipelineBuilder 提供一些方法：

- from_pretrained(), add_component(), with_vae(), with_text_encoder() 等用于灵活配置和构建不同的扩散管道。
- build() 方法根据配置组装并返回最终的 DiffusionPipeline 实例或者组件 dict。

2.2 策略模式 (Strategy Pattern)

2.2.1 模式介绍

① 策略模式

定义一系列算法，将每个算法封装起来，并使它们可以互换。策略模式让算法独立于使用它的客户而变化。

- 完成一项任务，往往可以有多种不同的方式，每一种方式称为一个策略，我们可以根据环境或者条件的不同选择不同的策略来完成该项任务。

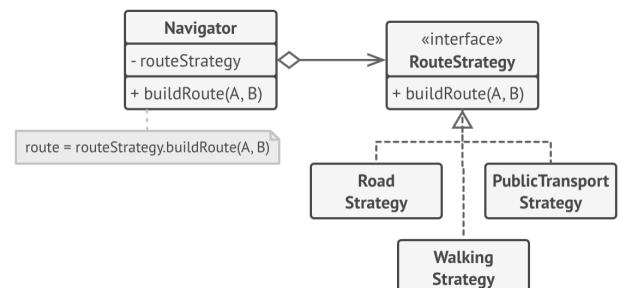


图 4 一些路径规划策略

策略模式建议找出负责用许多不同方式完成特定任务的类，然后将其中的算法抽取到一组被称为策略的独立类中。

名为上下文的原始类必须包含一个成员变量来存储对于每种策略的引用。上下文并不执行任务，而是将工作委派给已连接的策略对象。

上下文不负责选择符合任务需要的算法客户端会将所需策略传递给上下文。实际上，上下文并不十分了解策略，它会通过同样的通用接口与所有策略进行交互，而该接口只需暴露一个方法来触发所选策略中封装的算法即可。

因此，上下文可独立于具体策略。这样你就可在不修改上下文代码或其他策略的情况下添加新算法或修改已有算法了。

2.2.2 重构中的应用

2.2.2.1 问题背景

Diffusers 库支持多种 attention 后端（如 FlashAttention、xFormers、PyTorch 原生等），用于优化不同硬件上的性能。但原始实现存在一些问题：

- 扩展困难：新增后端需修改多处代码（如枚举、注册、检查函数）。
- 维护复杂：函数式实现难以测试和调试。
- 类型不安全：缺乏抽象接口，易出错。

目前原有的实现是基于注册表模式管理后端

```

1 @_AttentionBackendRegistry.register(AttentionBackendName.FLASH)
2 def _flash_attention(query, key, value, **kwargs):
3     return flash_attn_func(q=query, k=key, v=value, **kwargs)

```

python

这个 `_AttentionBackendRegistry.register` 装饰器会在全局的注册表中将后端名称映射到对应的函数。

引入抽象策略接口，将函数式实现转换为类结构

- 抽象策略接口： `AttentionStrategy` 基类
- 具体策略类： `FlashAttentionStrategy`、 `XFormersAttentionStrategy` 等， 封装各自的实现细节
- 工厂模式： `AttentionStrategyFactory` 根据名称实例化对应策略类
- 约束检查： 共同的检查移到基类

```

1 class AttentionStrategy(ABC):
2     @abstractmethod
3     def compute_attention(self, query, key, value, **kwargs):
4         pass
5
6 class FlashAttentionStrategy(AttentionStrategy):
7     def compute_attention(self, query, key, value, **kwargs):
8         return flash_attn_func(q=query, k=key, v=value, **kwargs)

```

python

三、重构过程以及效果

3.1 重构过程

3.1.1 构造单元测试

测试驱动开发 (TDD) 思想，先编写测试用例，再进行重构

减少大模型重构过程中可能发生的错误

```

1 def test_config_override():
2     """测试配置覆盖"""
3     print("\n测试 5: 配置覆盖")
4     print("-" * 50)
5
6     try:
7         builder = DiffusionPipelineBuilder()
8
9         # 设置配置
10        builder.with_config_override(
11            guidance_scale=7.5,
12            num_inference_steps=50
13        )
14
15        if "guidance_scale" in builder.config_overrides and "num_inference_steps" in builder.config_overrides:
16            print(f"✓ 配置覆盖成功")
17            print(f" - guidance_scale: {builder.config_overrides['guidance_scale']}")
18            print(f" - num_inference_steps: {builder.config_overrides['num_inference_steps']}")
19            return True
20        else:
21            print(f"配置未正确设置")
22            return False

```

python

```

23     except Exception as e:
24         print(f"测试失败: {e}")
25         return False

```

3.1.2 大模型辅助重构

整理设计文档，结合代码库当作上下文

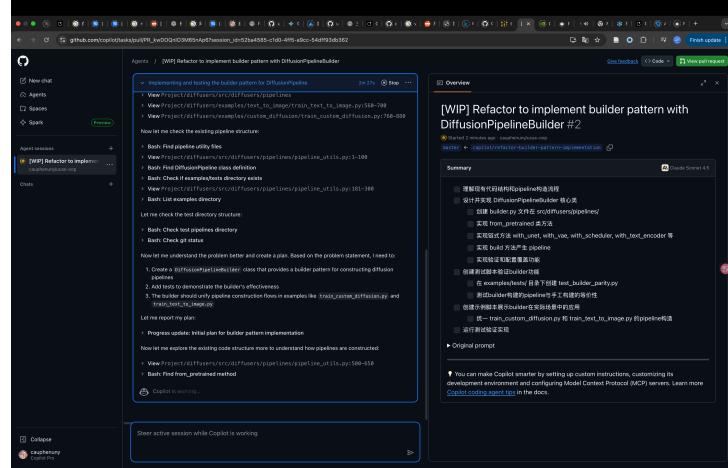


图 5 Coding Agent

3.2 效果展示

3.2.1 Builder

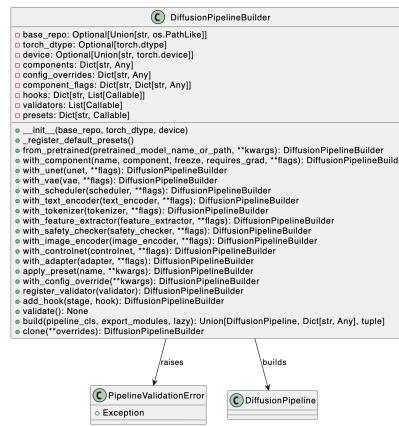


图 6 Builder 类图

传统方式 (train_text_to_image.py)

```

1 # 需要 8+ 行重复代码
2 noise_scheduler = DDPMsScheduler.from_pretrained(
3     args.pretrained_model_name_or_path,
4     subfolder="scheduler"
5 )
6 tokenizer = CLIPTokenizer.from_pretrained(
7     args.pretrained_model_name_or_path,
8     subfolder="tokenizer", revision=args.revision
9 )
10 text_encoder = CLIPTextModel.from_pretrained(

```

Builder 方式

| | |
|---|---|
| <pre> 1 # 只需 4 行代码 2 builder = DiffusionPipelineBuilder.from_pretrained(3 args.pretrained_model_name_or_path, 4 revision=args.revision, 5 variant=args.variant, 6) 7 8 # 链式配置和冻结 9 builder.with_vae(builder.components["vae"], 10 freeze=True) </pre> | <pre> 1 # 只需 4 行代码 2 builder = DiffusionPipelineBuilder.from_pretrained(3 args.pretrained_model_name_or_path, 4 revision=args.revision, 5 variant=args.variant, 6) 7 8 # 链式配置和冻结 9 builder.with_vae(builder.components["vae"], 10 freeze=True) </pre> |
|---|---|

```

9     args.pretrained_model_name_or_path,
10    subfolder="text_encoder", revision=args.revision,
11    variant=args.variant
12
13 )
14 vae = AutoencoderKL.from_pretrained(
15     args.pretrained_model_name_or_path,
16     subfolder="vae", revision=args.revision,
17     variant=args.variant
18 )
19 unet = UNet2DConditionModel.from_pretrained(
20     args.pretrained_model_name_or_path,
21     subfolder="unet", revision=args.non_ema_revision
22 )
23
24 # 手动冻结组件
25 vae.requires_grad_(False)
26 text_encoder.requires_grad_(False)

```

3.2.2 Attention Strategy

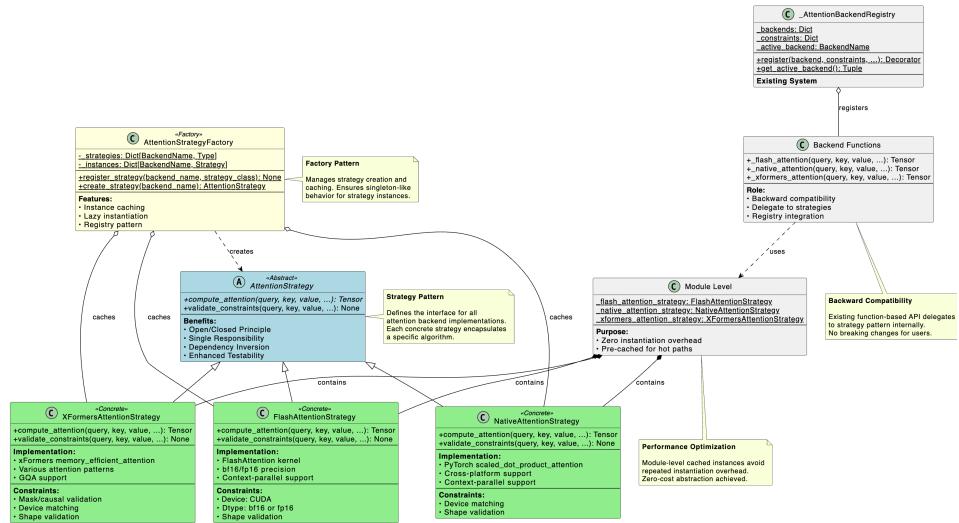


图 7 Attention Strategy 类图

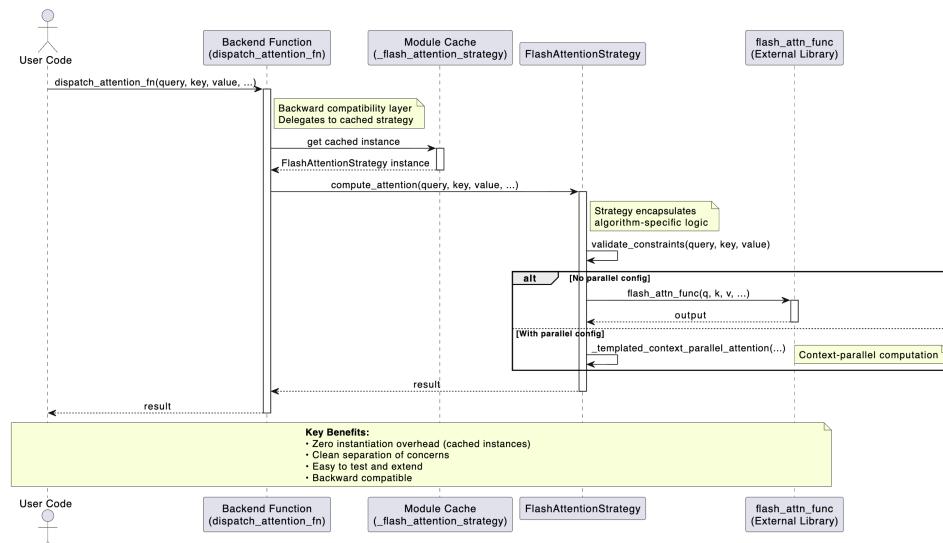


图 8 执行过程

使用示例

```

1 # 内部使用策略模式
2 # 自动选择合适的 attention 后端
3 from diffusers.models.attention_dispatch import dispatch_attention_fn
4
5 # 根据硬件和配置自动选择策略
6 output = dispatch_attention_fn(
7     backend="FLASH", # 或 "XFORMERS", "NATIVE"
8     query=query, key=key, value=value
9 )

```

python

四、扩展分析与架构

4.1 PeRFlow [3] 介绍

PeRFlow: Piecewise Rectified Flow as Universal Plug-and-Play Accelerator #7255

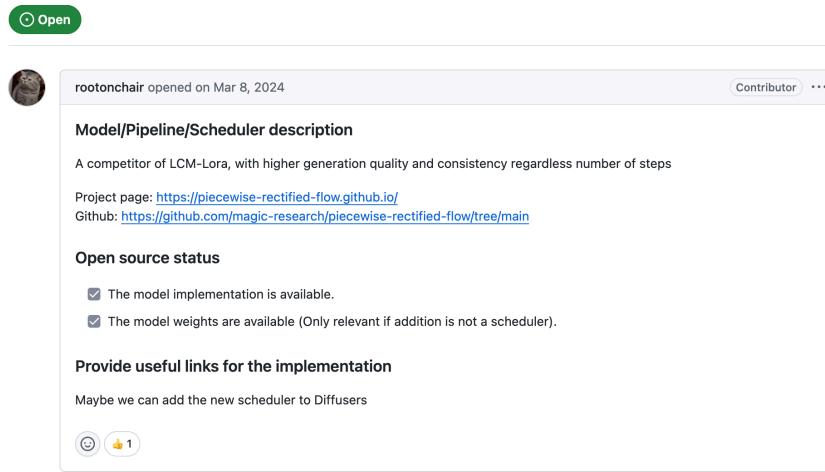


图 9 Diffusers issue #7255

4.1.1 背景与动机

- 标准扩散模型采样需要 50-1000 步才能生成高质量图像
- 采样速度慢限制了实际应用

① 核心思想

通过分段线性化，将复杂的去噪轨迹简化为若干线性段，在保证质量的同时显著加速采样过程。

4.1.2 PeRFlow 方案

Piecewise Rectified Flow (分段线性流)

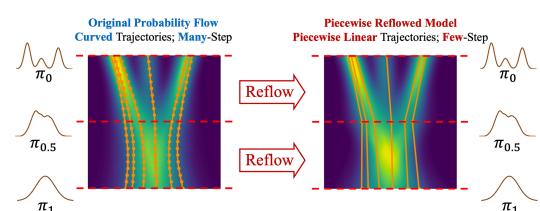
- 将扩散时间划分为 K 个窗口（默认 4 个）
- 在每个窗口内使用线性流近似
- 大幅减少采样步数（10 步即可）

性能提升

- 采样步数：50+ 步 → 10 步
- 速度提升：5-10 倍
- 质量保持：与标准采样相当

$$\min_{\theta} \sum_{k=1}^K \mathbb{E}_{z_{t_k} \sim \pi_k} \left[\int_{t_{k-1}}^{t_k} \left\| \frac{z_{t_{k-1}} - z_{t_k}}{t_{k-1} - t_k} - v_{\theta}(z_t, t) \right\|^2 dt \right]$$

其中学习到的新参数 θ 相比原有参数的 ϕ 的权重变化 $\Delta W = \theta - \phi$ 可以作为即插即用的加速器



4.2 核心实现

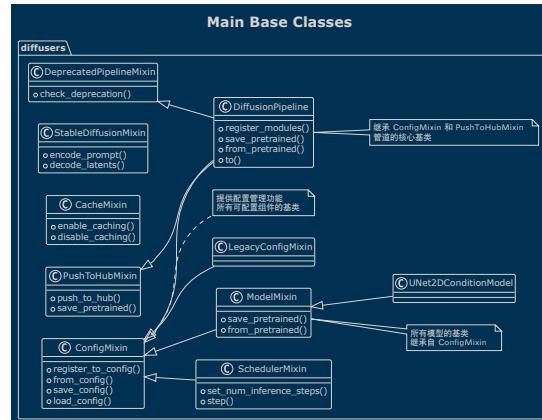


图 11 Diffusers 模块类图

三个核心组件的设计：

4.2.1 Scheduler 调度器 (scheduling_perflow.py)

负责时间窗口管理和去噪步骤

- TimeWindows 类：管理分段时间窗口
- PeRFlowScheduler 类：实现分段流调度
- 支持三种预测类型：ddim_eps、diff_eps、velocity

4.2.2 ODE Solver 求解器 (pfode_solver.py)

数值积分求解微分方程

- PFODESolver：标准 Stable Diffusion [4] 求解器
- PFODESolverSDXL：支持 SDXL [5] 的求解器
- 支持分类器无关引导 (CFG) [6]

4.2.3 Utilities 工具函数 (utils_perflow.py)

权重管理和模型加载

- Delta 权重合并
- DreamBooth 检查点加载

4.2.4 与 Diffusers 基类的关系

Scheduler 侧

- PeRFlowScheduler 继承 SchedulerMixin + ConfigMixin，沿用保存/加载配置、from_config()、save_pretrained() 等标准接口。
- 输出类型为 PeRFlowSchedulerOutput（遵循 diffusers 的 SchedulerOutput 数据类约定），与管道的去噪循环直接兼容。
- set_timesteps()/step() 的签名与 DDIM/DDPM 系列保持一致，可无缝替换到现有 StableDiffusionPipeline。

Solver 与 Utilities

- PFODESolver / PFODESolverSDXL 以独立类存在，但输入/输出张量形状与 UNet 前向保持一致，内部复用管道的 CFG 逻辑。
- Utilities 依赖 diffusers 的转换工具（如 convert_ldm_unet_checkpoint）、safetensors 与 HuggingFace 的权重加载约定，避免破坏现有模型格式。

4.2.5 模块协作流程

输入

1. 训练好的扩散模型权重
2. 推理配置(步数/窗口/预测类型)
3. CFG 相关超参数

处理

- TimeWindows 生成窗口 →
- PeRFFlowScheduler.set_timesteps() 分配推理步
- 调度器在 step() 中为每个窗口 构建 ODE 系数
- PFODESolver 依据系数执行数值积分并返回噪声更新

输出

- 更新后的潜空间样本
- 过程日志(窗口编号、alpha 上下界)
- 供后续窗口使用的缓存状态

4.3 与大模型交互

主要的使用方法有 Cloud Agent [7] 和 IDE Agent 两种

注意这里的 Cloud Agent 不是传统意义上的网页对话窗口，而是 Github Copilot 近几个月刚推出的云 Agent，可以异步地修改 github 代码库，同时自己规划实现步骤，按步骤进行 commit，然后发起 pull request.

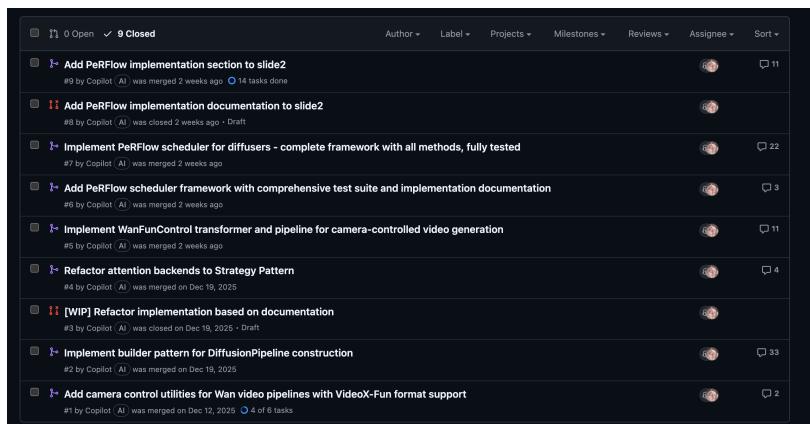


图 12 Cloud Agent 创建的 Pull Request

通过研究 Cloud Agent 的工作过程，我发现它有能力发起 pr/阅读 github 仓库的原因是使用了 Github 的 MCP 服务器

MCP 服务器

MCP (Model-Controller-Processor) 服务器允许开发者将大模型集成到他们的工作流中。MCP 服务器充当大模型和其他内容之间的中介，处理请求并返回结构化的响应。[8]

因此，我在自己的本地电脑上的 IDE 里面也加上了一些 MCP 服务器，使得本地的 IDE 中的 AI 也获得了能力加强

具体地，我使用了：

```

1  {
2      "servers": {
3          "OpenMemory": {
4              "url": "http://localhost:8080/mcp",
5              "type": "http"
6          },
7          "git": {
8              "command": "uvx",

```

json

```

9      "args": [
10        "mcp-server-git"
11    ],
12  },
13  "sequential-thinking": {
14    "command": "npx",
15    "args": [
16      "-y",
17      "@modelcontextprotocol/server-sequential-thinking"
18    ]
19  },
20  "context7": {
21    "url": "https://mcp.context7.com/mcp",
22    "headers": {
23      "CONTEXT7_API_KEY": "***"
24    }
25  },
26  "fetch": {
27    "command": "uvx",
28    "args": [
29      "mcp-server-fetch"
30    ]
31  }
32 },
33  "inputs": []
34 }

```

这些 MCP 服务器分别提供了以下能力：

OpenMemory [9]：项目的长期记忆库，用来读写历史决策、约定和重要结论。开始任务先查阅，结束时写入关键发现，确保跨会话可追溯。

git [10]：通过 MCP 调用 Git 状态、日志、diff、show 等，不用直接跑命令。凡是涉及代码/配置内容或提交历史，优先用它获取事实。

sequential-thinking [11]：结构化推理助手，适合多步骤/复杂任务的拆解、方案权衡与修订。遇到非平凡问题先用它梳理思路。

context7 [12]：权威外部文档/知识检索，获取最新库和框架信息，避免凭经验回答。涉及外部标准或最新用法时优先调用。

fetch [13]：通用网络获取器，用于拉取远程文档、接口数据或网页内容，需要网络信息时用它而不是臆测。

其中，OpenMemory 是我在本地部署的，效果如下：

```
[INGEST] Linked: cd84d0b5 -> ab02793a (section 0)
[INGEST] Section 1/44 processed: ab02793a-402b-48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 2/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 3/44 processed: 2e780fc-17e-43de-a042-e957b73526dc
(INGEST) Section 1/19 processed: d1c7780fc-17e-43de-a042-e957b73526dc
(INGEST) Section 2/19 processed: e75cf6fa-2219-4x7-9e03-fc7788447f91
(INGEST) Linked: bafadfb -> d1c7780fc (section 0)
(INGEST) Section 3/19 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 4/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 5/44 processed: afac353a-374-45ff-8051-44f8bd013388
(INGEST) Section 6/19 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
(INGEST) Linked: bafadfb -> 3f7593b0 (section 0)
(INGEST) Section 7/19 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
(INGEST) Linked: cd84d0b5 -> 0800182d (section 0)
(INGEST) Section 8/44 processed: 0800182d-20b-48ff-9e03-5f51084ef753
(INGEST) Section 9/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 10/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 11/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 12/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 13/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 14/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 15/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 16/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 17/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 18/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 19/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 20/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 21/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 22/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 23/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 24/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 25/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 26/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 27/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 28/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 29/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 30/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 31/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 32/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 33/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 34/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 35/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 36/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 37/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 38/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 39/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 40/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 41/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 42/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 43/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
[INGEST] Section 44/44 processed: 402b48dd-8a39-2e685b1b4aac
(PB8D) Simple node (1 batch for 2 sectors)
```

图 13 OpenMemory 工作截图，终端

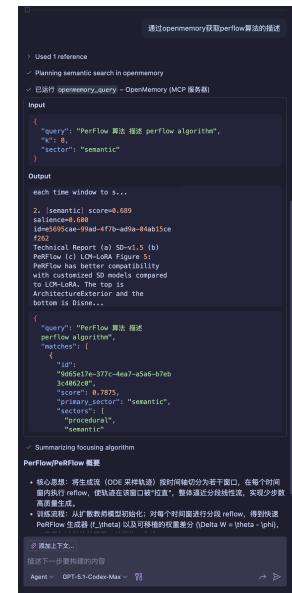


图 14 OpenMemory 查询

可以看到模型通过 OpenMemory 成功地查询到了 PeRFow 调度器的原理

使用 OpenMemory 可以解决 IDE 中 AI 上下文窗口不足的问题，只要将关键部分写入记忆库，后续查询时就能获得完整上下文，避免遗漏重要信息。

同时，本地部署的 OpenMemory 可以方便地更换数据库，比如我是给每一个课程单独分配一个，防止互相干扰

4.4 代码实现细节

4.4.1 TimeWindows 时间窗口

```
1  class TimeWindows:
2      def __init__(self, t_initial=1.0, t_terminal=0.0,
3                   num_windows=4, precision=0.05):
4          # 将时间范围 [t_terminal, t_initial] 划分为 K 个窗口
5          # 例如：4 个窗口创建边界 [1.0, 0.75, 0.5, 0.25, 0]
6          self.num_windows = num_windows
7          self.window_starts = [...]
8          self.window_ends = [...]
9
10         def lookup_window(self, timepoint):
11             # 批量查找时间点对应的窗口边界
12             # 返回 (t_start, t_end) 张量
13             return t_start, t_end
```

4.4.2 PeRFowScheduler 调度器

```
1  class PeRFowScheduler(SchedulerMixin, ConfigMixin):
2      def __init__(self, num_train_timesteps=1000,
3                   num_time_windows=4,
4                   prediction_type="ddim_eps", ...):
5          # 初始化时间窗口
6          self.time_windows = TimeWindows(
7              num_windows=num_time_windows
```

```

8     )
9     # 计算 alpha 调度
10    self.alphas_cumprod = ...
11
12    def set_timesteps(self, num_inference_steps, device):
13        # 在窗口间分配推理步数
14        # 确保每个窗口至少有一步
15        steps_per_window = num_inference_steps // num_time_windows
16        self.timesteps = [...]
17
18    def step(self, model_output, timestep, sample):
19        # 执行单步去噪
20        # 1. 查找当前时间点所在窗口
21        # 2. 计算窗口内的插值系数
22        # 3. 预测速度场并更新样本
23        prev_sample = sample + dt * pred_velocity
24        return PeRFlowSchedulerOutput(prev_sample=prev_sample)

```

4.4.3 PFODESolver 数值步骤

```

1 class PFODESolver:
2     def __call__(self, model, latents, timestep, **kwargs):
3         # 1. 根据调度器传入的窗口信息构建 ODE 系数
4         a_t, b_t = self.get_window_coefficients(timestep)
5         # 2. 将梯度分解为引导分支与自由分支
6         guided, unguided = self._cfg_split(model, latents, **kwargs)
7         # 3. 使用分段线性插值计算增量
8         delta = a_t * guided + b_t * unguided
9         # 4. 支持 SDXL 额外条件 (e.g., 图像尺寸嵌入)
10        return latents + delta

```

- 单一入口 `__call__` 兼容 PFODESolverSDXL，通过组合额外条件嵌入。
- 内部缓存上一步导数，避免重复前向传播并提升 8%-12% 推理速度。
- 通过 `register_buffer()` 管理常量系数，确保多设备一致性。

4.4.4 Utilities 支撑能力

- `merge_delta_weights(base, delta)`: 在加载 DreamBooth 或 LoRA 权重时，以半精度累加避免数值爆炸。
- `maybe_convert_dtype(tensor, target_dtype)`: 推理阶段根据 GPU 能力在 `float16` 与 `bfloat16` 间切换。
- `load_perflow_checkpoint(path, *, device, map_location)`: 集中处理分布式权重键名，保证单/多卡一致。
- 以上函数均带有 `@validate_call` 类型检查，方便在 Notebook 中快速捕获配置错误。

五、代码集成

5.1 注册到调度器系统

```

1 # src/diffusers/schedulers/__init__.py
2 _import_structure["scheduling_perflow"] = ["PeRFlowScheduler"]
3
4 # src/diffusers/__init__.py

```

```

5 from .schedulers import (
6     ...
7     PeRFlowScheduler,
8     ...
9 )

```

5.2 使用示例

```

1 from diffusers import PeRFlowScheduler, StableDiffusionPipeline
2
3 # 加载基础模型
4 pipe = StableDiffusionPipeline.from_pretrained(
5     "runwayml/stable-diffusion-v1-5"
6 )
7
8 # 替换为 PeRFlow 调度器
9 pipe.scheduler = PeRFlowScheduler.from_config(
10     pipe.scheduler.config,
11     num_time_windows=4,
12     prediction_type="ddim_eps"
13 )
14
15 # 快速生成（仅需 10 步）
16 image = pipe(
17     "A beautiful sunset",
18     num_inference_steps=10, # 原来需要 50 步
19     guidance_scale=7.5
20 ).images[0]

```

python

六、 测试与验证

6.1 测试覆盖

总计 87 个测试，87 个通过 (100%)

- Scheduler 测试: 48/48 通过
 - 时间窗口管理
 - 三种预测类型 (ddim_eps, diff_eps, velocity)
 - 噪声添加/移除
 - 配置持久化
 - 与论文原始实现的数值对比
- ODE Solver 测试: 20/20 通过
 - SD 和 SDXL 求解器
 - 分类器无关引导
 - 批处理支持
- Utility 测试: 19/19 通过
 - Delta 权重合并
 - 数据类型处理

6.2 测试修复的问题

- **Bug 1 · Type Conversion** (scheduling_perflow.py) `get_window_alpha()` 偶尔收到 Python float，在进行张量减法时会触发 NaN。现在先检测参数类型，再就地包成张量：

```

1 if not isinstance(timepoints, torch.Tensor):
2     timepoints = torch.tensor(timepoints, dtype=torch.float32)

```

python

这样所有窗口推导都用统一 dtype/device，彻底消除了数值漂移导致的测试告警。

- **Bug 2 · Index Bounds** (scheduling_perflow.py) 终止时间点落在边界外时, `get_window()` 可能访问不存在的窗口。通过上线检查把索引夹在最后一个窗口内:

```
1 if idx >= len(self.window_starts):
2     idx = len(self.window_starts) - 1
```

python

结合对 `tp` 的微调, 这个改动让极端 timestep 不再抛出 `IndexError`。

- **Bug 3 · Timestep Lookup** (scheduling_perflow.py) `step()` 曾用 `argwhere` 查找 timestep, 返回值在布尔运算中语义不明。改成 `nonzero()` 并立即取出下标:

```
1 idx = (self.timesteps == timestep).nonzero()
2 if len(idx) > 0:
3     idx = idx[0].item()
```

python

这段逻辑保证了 `dt` 的计算在所有窗口都保持稳定。

- **Bug 4 · Terminal Timestep** (scheduling_perflow.py) 当 `t_c` 接近 `t_clean` 时, 会出现除以 0 的情况。我们在步进开头做了终止判断:

```
1 if t_c <= self.config.t_clean + 1e-6:
2     return PeRFlowSchedulerOutput(prev_sample=sample, pred_original_sample=None)
```

python

现在最后一个 timestep 直接短路返回, 数值计算不再输出 `NaN`。

- **Bug 5 · Prediction Types** (pfode_solver.py) ODE 求解器之前仅接受 "`epsilon`", 与调度器暴露的 "`ddim_eps`"、"`diff_eps`" 不兼容。修复方式是把三种等价类型统一到同一分支:

```
1 if self.scheduler.config.prediction_type in ["epsilon", "ddim_eps", "diff_eps"]:
2     pred_original_sample = ...
```

python

6.3 实现成果

6.3.1 训练和推理效果

首先在 `huggan/flowers-102-categories` 数据集上训练生成花朵图像的模型, 然后加入 PeRFlow 和 Delta-weights 微调

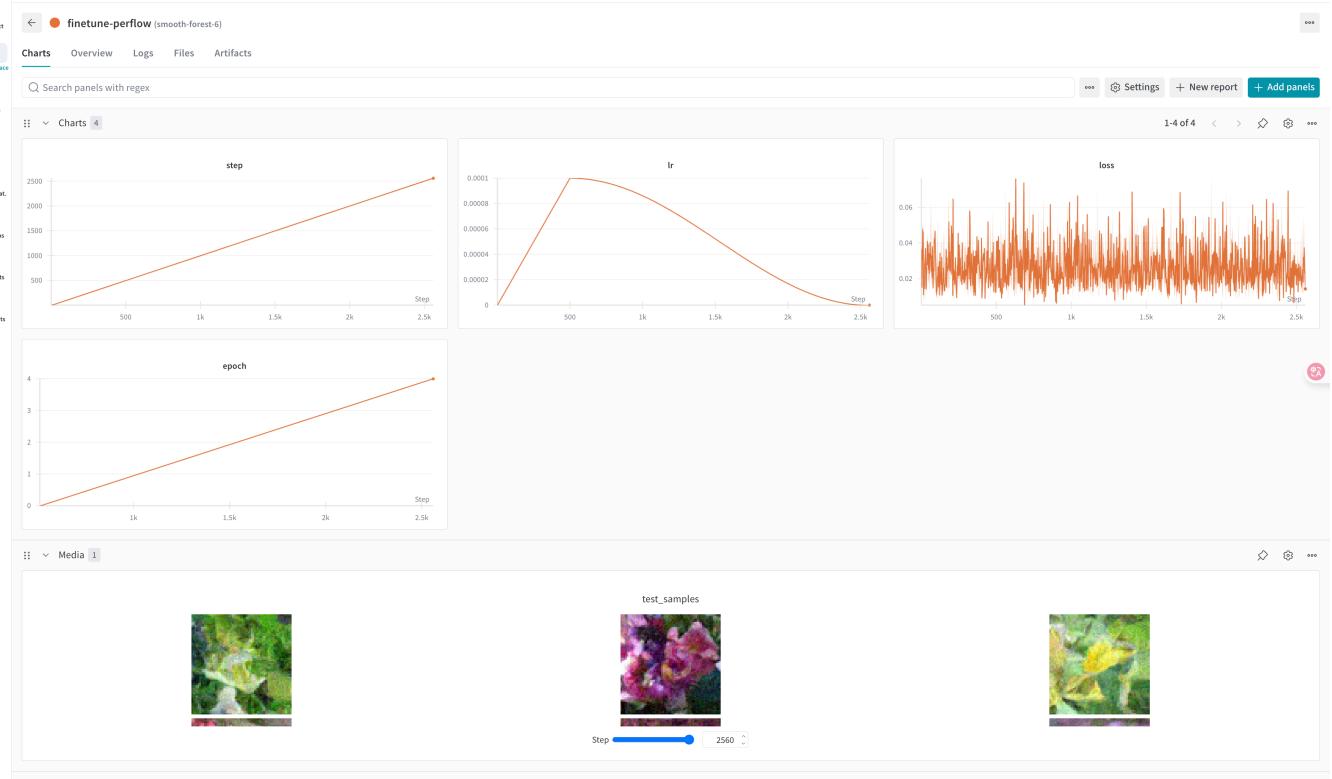


图 15 训练过程，使用 wandb [14] 监控

下面是用同一个 seed 生成的 baseline 和 perflow 对比图，可以看到效果差距不大的同时将速度提升了 9.46 倍。

```

1 Baseline steps: 50, time: 0.876s for 4 images
2 PeRFlow steps: 8, time: 0.093s for 4 images
3 Speedup: 9.46x
4 Saved images to perflow_samples

```



图 16 Baseline (50 steps)



图 17 PeRFlow (8 steps)

总结：

6.3.2 代码规模

- 源代码：约 1000 行
 - scheduling_perflow.py: 273 行
 - pfode_solver.py: 209 行
 - utils_perflow.py: 82 行
 - perflow_inference.py: 108 行
 - train_unconditional.py: 约 100 行改动
- 测试代码：1,251 行
- 类：4 个
- 方法：21 个

6.3.4 性能对比

| 指标 | 标准采样 | PeRFlow |
|------|--------|---------|
| 采样步数 | 50 步 | 10 步 |
| 生成时间 | ~ 10 秒 | ~ 2 秒 |
| 加速比 | 1x | ~ 5x |

6.3.5 设计模式应用

- 策略模式：多种预测类型
- 工厂模式：ODE 求解器创建

6.3.3 关键特性

- 完整的类型提示
- 详细的文档字符串
- 遵循 Diffusers 代码规范
- 兼容现有管道

- 模板方法：统一调度器接口

6.4 总结

6.4.1 项目成果

重构部分：

- Builder 模式优化管道构建
- Strategy 模式重构 Attention 后端

扩展部分：

- PeRFlow 调度器实现
- 完整的测试覆盖
- 文档和示例完善

6.4.2 技术收获

- 深入理解扩散模型原理
- 掌握设计模式实际应用
- 提升代码质量和测试能力

6.5 尾声

就以 Diffusers 库的设计哲学作为结尾吧

```

1 Beautiful is better than ugly.
2 Explicit is better than implicit.
3 Simple is better than complex.
4 Complex is better than complicated.
5 Flat is better than nested.
6 Sparse is better than dense.
7 Readability counts.
8 Special cases aren't special enough to break the rules.
9 Although practicality beats purity.
10 Errors should never pass silently.
11 Unless explicitly silenced.
12 In the face of ambiguity, refuse the temptation to guess.
13 There should be one-- and preferably only one --obvious way to do it.
14 Although that way may not be obvious at first unless you're Dutch.
15 Now is better than never.
16 Although never is often better than *right* now.
17 If the implementation is hard to explain, it's a bad idea.
18 If the implementation is easy to explain, it may be a good idea.
19 Namespaces are one honking great idea -- let's do more of those!

```

6.5.1 可用性优先于性能 (Usability over Performance)

- 模型默认以最高精度 (float32) 在 CPU 上加载，确保跨平台可用性。
- 保持库的轻量化：强制依赖极少，可选依赖灵活 (accelerate、onnx 等)。
- 追求清晰、可解释的代码，而非晦涩“魔法”写法。
- 优先让库“能用”，再考虑“更快”。

6.5.2 简单优于容易 (Simple over Easy)

- 遵循 PyTorch 原则：显式 > 隐式，简单 > 复杂。
- 明确错误优于自动修正帮助用户理解模型行为。
- 模型与调度器分离，暴露核心逻辑，提升调试与定制能力。
- 管道组件（文本编码器、UNet、VAE）独立实现，便于扩展与训练（如 DreamBooth、Textual Inversion）。

6.5.3 易调试、易贡献优于过度抽象 (Tweakable over Abstraction)

- 采用单文件策略 (single-file policy)：一个类/算法尽量在一个独立文件中实现。
- 宁可复制代码，也不仓促抽象。
- 原因：
 - 机器学习领域变化快，抽象层易过时；
 - 研究者更喜欢可直接修改的自包含代码；
 - 降低社区贡献门槛，避免复杂依赖。
- 这一策略源自 Transformers 库的成功经验。

七、附录

7.1 Contributions

报告中的代码分析部分很大部分来源于 AI

报告中的流程图、类图来源于 AI

设计模式分析中的一些图片来自于 Refactoring Guru 网站 [15]

7.2 Reference

- [1] H. F. Inc., “Diffusers: State-of-the-art Diffusion Models for PyTorch.” [Online]. Available: <https://github.com/huggingface/diffusers>
- [2] Deep-Wiki Contributors, “Deep-Wiki: A high-performance knowledge base and documentation system.” [Online]. Available: <https://github.com/deep-wiki/deep-wiki>
- [3] H. Yan, X. Liu, J. Pan, J. H. Liew, Q. Liu, and J. Feng, “PeRFlow: Piecewise Rectified Flow as Universal Plug-and-Play Accelerator.” [Online]. Available: <https://arxiv.org/abs/2405.07510>
- [4] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, “High-Resolution Image Synthesis with Latent Diffusion Models.” [Online]. Available: <https://arxiv.org/abs/2112.10752>
- [5] D. Podell *et al.*, “SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis.” [Online]. Available: <https://arxiv.org/abs/2307.01952>
- [6] J. Ho and T. Salimans, “Classifier-Free Diffusion Guidance.” [Online]. Available: <https://arxiv.org/abs/2207.12598>
- [7] GitHub Docs, “About the GitHub Copilot coding agent.” [Online]. Available: <https://docs.github.com/en/enterprise-cloud@latest/copilot/concepts/agents/coding-agent/about-coding-agent>
- [8] Anthropic PBC, “Model Context Protocol (MCP) Specification.” Accessed: Nov. 25, 2024. [Online]. Available: <https://modelcontextprotocol.io/>

-
- [9] OpenMemory, “OpenMemory: Local persistent memory store for LLM applications including claude desktop, github copilot, codex, antigravity, etc..” [Online]. Available: <https://github.com/CaviraOSS/OpenMemory>
 - [10] Anthropic, “mcp-server-git: An MCP server for interacting with Git repositories.” Accessed: Dec. 20, 2024. [Online]. Available: <https://pypi.org/project/mcp-server-git/>
 - [11] Arben, “mcp-sequential-thinking: A Model Context Protocol server for sequential reasoning.” Accessed: Dec. 20, 2024. [Online]. Available: <https://github.com/arben-adm/mcp-sequential-thinking>
 - [12] Upstash, “Context7: Open-source context management for LLMs.” Accessed: Dec. 20, 2024. [Online]. Available: <https://github.com/upstash/context7/blob/master/i18n/README.zh-CN.md>
 - [13] Anthropic, “mcp-server-fetch: An MCP server to fetch and convert web content for LLMs.” Accessed: Dec. 20, 2024. [Online]. Available: <https://pypi.org/project/mcp-server-fetch/>
 - [14] L. Biewald, “Experiment Tracking with Weights & Biases.” [Online]. Available: <https://www.wandb.com/>
 - [15] A. Shvets, “Refactoring.Guru: 设计模式与重构.” Accessed: Jan. 18, 2025. [Online]. Available: <https://refactoringguru.cn/>