Typed ModuleSpec

Date: Nov 29, 2025
Author: Nicolas Schank, ReflectionAI (nschank)
Self Link: go/typed-modulespec-dd (ReflectionAI-internal)
Reviewers: Megatron-LM Maintainers
This document is publicly available for viewing. For comment access, reach out to the author.

Summary

The document proposes replacing Megatron-LM's opaque `ModuleSpec` configuration pattern with a type-safe system based on Python `Protocol` and `ParamSpec`. This change will significantly improve code readability and maintainability by enabling static type-checking for module configuration, instantiation, and usage, thereby reducing the probability of bugs and easing API refactoring.

Background

ModuleSpec

Megatron-LM is an Nvidia framework for training transformer models at scale, with an emphasis on modularity and flexibility. The core Megatron library has very few required dependencies and avoids making assumptions about the structure of the model being trained - instead, it uses a comprehensive configuration to construct and train an arbitrary model, which has a number of upsides:

- **Flexibility**: The same code can be used for models with totally different structures (e.g. small dense models vs. enormous MoE models), simply by altering the configuration.
- **Modularity**: Different building blocks can be swapped in when more esoteric/advanced libraries are available, allowing Megatron to *leverage* cutting-edge optimizations without needing to add them as required dependencies for all users.
- **Customizability**: Power users can swap in their own implementations of modules if desired.
- **Ease of Use:** Megatron comes with a 'toolbox' of standard modules out of the box, to support the construction of well-known models (like Llama, Qwen, DeepSeek, etc.) easily.

As a pytorch-based framework, a key requirement of Megatron's configuration library is the ability to construct `torch.nn.Module` instances on the fly. The current primary interface for doing so is a class called `ModuleSpec`[1]:

```python
@dataclass
class ModuleSpec:
    module: Union[Tuple, type]
    params: dict = field(default_factory=lambda: {})
    submodules: type = None
```

This object provides a way of specifying how a particular torch module should be constructed during model instantiation, with a few built-in features of note:

- `module` can either be a particular type of `nn.Module`, or a tuple which provides the *location* of a particular module to import and construct.
- `params` provides a way to pass individual keyword args which will be provided to the Module's initializer.
- `submodules` optionally acts as an additional keyword arg (also named `submodules`) which will be passed to the `nn.Module`'s initializer in addition to `params`; this reflects a common pattern used by many built-in `nn.Module` instances provided by Megatron (described in more detail below).

`ModuleSpec` instances are instantiated at model-construction time using the method `build_module`[2], which is intentionally unopinionated about the shape of its input in order to provide additional flexibility:

```python
def build_module(spec_or_module: Union[ModuleSpec, type], *args, **kwargs):
```

In particular, the provided `spec_or_module` instance can be any of the following:

- A function or lambda, in which case it is returned unchanged and `args`/`kwargs` are unused.[3]

---

[1] Captured at this commit; docstring omitted for brevity, type hints and defaults preserved as-is. Note that the type hint for `submodules` is incorrect and should be `object`.

[2] Signature copied from this commit; type hints preserved as-is. The full signature should in theory be `build_module(spec_or_module: Union[ModuleSpec, types.FunctionType, type], *args: object, **kwargs: object) -> object`.

[3] This allows for 'dropping in' arbitrary functions to be used in the place of a `nn.Module` instance.

    - *Note*: Other callables - such as types, classes with a `__call__` method, or functions wrapped in `functools.partial` - are not captured in this case.
- A type (say, `Foo`) in which case `build_module` returns `Foo(*args, **kwargs)`.
- A `ModuleSpec`, in which case the appropriate module is constructed; the `args`/`kwargs` provided to `build_module` are merged with the ones specified in the `ModuleSpec` itself.

Individual Modules in Megatron attempt to standardize their use alongside `ModuleSpec`, specifically by using the following pattern[4]:

```python
@dataclass
class SelfAttentionSubmodules:
  linear_qkv: Union[ModuleSpec, type] = None
  core_attention: Union[ModuleSpec, type] = None
  linear_proj: Union[ModuleSpec, type] = None
  # ...

class SelfAttention(MegatronModule):
  def __init__(
    self,
    config: TransformerConfig,
    submodules: SelfAttentionSubmodules,
    layer_number: int,
    attn_mask_type=AttnMaskType.padding,
    # ...
  ):
    self.linear_qkv = build_module(
      submodules.linear_qkv, config, config.hidden_size, ...
    )
    # ...

  def forward(...):
    # ...
    mixed_qkv, _ = self.linear_qkv(hidden_states)
    # ...
```

To summarize, modules will declare the submodules they expect to build as a dataclass, then accept an instance of that dataclass as a `submodules` argument in their initializer (to be provided as part of the `ModuleSpec`). Then, the module constructs those submodules using `build_module`, with most arguments being provided explicitly (rather than as part

[4] Adapted from the SelfAttention module at this commit, but simplified and snipped for clarity.

of `ModuleSpec`). Finally, the constructed module will be used directly via its `__call__` interface (and, occasionally, other methods may be used as well).

The actual configuration of a module then looks something like this[5]:

```python
return ModuleSpec(
  module=TransformerLayer,
  submodules=TransformerLayerSubmodules(
    self_attention=ModuleSpec(
      module=SelfAttention,
      params={"attn_mask_type": AttnMaskType.causal},
      submodules=SelfAttentionSubmodules(
        linear_qkv=backend.column_parallel_layer_norm_linear(),
        core_attention=backend.core_attention(),
        linear_proj=backend.row_parallel_linear(),
        ...))))
```

Note that only the two arguments `attn_mask_type` and `submodules` are set by this `ModuleSpec` instance; the other arguments to `SelfAttention` are instead meant to be provided by its outer module (in this case, `TransformerLayer` when it constructs its `self_attention` submodule).

Limitations of ModuleSpec

Despite the flexibility and modularity it provides, the current design of `ModuleSpec` is awkward and prone to errors due to its opacity and its incompatibility with type-checking.

There are three 'steps' that take place when configuring and instantiating these models, and at each one the design of `ModuleSpec` misses a few opportunities to improve clarity and reduce the probability of bugs:

1. At **configuration time** (when building `ModuleSpec` instances):
    - Constructing a `ModuleSpec` requires that callers know which parameters are being provided by the wrapping class, and which ones are meant to be provided by `params` instead. Distinguishing the `submodules` object helps a bit, but also adds confusion since it's purely a convention which is redundant with the `params` field in `ModuleSpec`.
    - All modules are constructed by `ModuleSpec`, so most of the configuration helper methods have the same return type and can only be distinguished by name.

[5] Snipped from this commit.

    - The indirection caused by `ModuleSpec` is confusing, and makes it difficult for someone unfamiliar with Megatron to look at a configuration and to parse what it will actually do.
    - The extreme flexibility of `build_module` also creates a lot of confusion, since it provides so many different means of doing the same thing, with subtly different results.
2. At **module construction time** (when creating a `torch.nn.Module` instance from a `ModuleSpec` using `build_module`):
    - `build_module` is completely opaque, so there is no type-checking to verify that the correct parameters are passed. This is particularly problematic since it is common to pass 5 or more parameters when constructing each module.
3. At **module usage time** (when using methods of the object returned by `build_module`):
    - `build_module` also *returns* an object with no hint as to its interface or behaviors, so there is no enforcement that the object is being used correctly.

Taken together, this makes it so that Megatron's core functionality is basically unable to benefit from type hints at all, which is an enormous readability and maintainability burden. It is very difficult to safely add or rename the parameters of a module's initializer, because it requires manual human effort to mentally 'unwrap' relevant `ModuleSpec` instances to understand what callsites might be affected and to fix them.

Worse, because `ModuleSpec` is meant to provide *external users* with a way to opaquely interact with the building blocks of Megatron, there is no way to easily 'communicate' such changes to them except by simply breaking them and letting them figure it out manually. The indirection and type-hostility of `ModuleSpec` turns trivial improvements to the API (like renaming or adding parameters) into huge inconveniences for users, since an upgrade to Megatron can silently invalidate their custom `Module` definitions or configurations.

Python Typing

Type hints, introduced in PEP 484, provide Python code with a means of declaring type signatures for functions without sacrificing the language's dynamic nature. Most importantly, type hints are optional and do not (by default) affect the runtime behavior of the code: they are simply there to improve code readability and maintainability through static analysis. Some key benefits are:

- **Documentation and Readability:** Type hints act as embedded, executable documentation, making the expected inputs and outputs of functions, methods, and variables immediately clear to other developers.
- **Static Analysis:** Static type checkers (like mypy and pyright) can analyze the codebase without needing to execute the code, and will identify the misuse of interfaces automatically.
- **Gradual/Soft Adoption:** Type-checking can be introduced incrementally to existing codebases, and type-checking failures are simply lint errors which do not affect code at runtime at all.
- **Easy Escape Hatches:** In exceptional or subtle situations, there are easy ways to 'force' the type checker to treat something a different way (e.g. by use of a `# type: ignore` comment or by using `cast`). This provides both flexibility and a means of self-documenting subtle code.

Basic type hinting requires little setup (since classes and built-ins like `ModuleSpec` and `int` can simply be added as type hints directly), but type hints for really complex interfaces (such as the ones serviced by `ModuleSpec`) requires some more advanced tooling.

Protocol

The `Protocol` type (introduced in PEP 544) formalizes the concept of **Structural Subtyping** or **Duck Typing**. In Python, "duck typing" means an object's validity is determined by what methods and attributes it *has*, not what class it *inherits* from.

A `Protocol` allows you to define an interface purely by specifying the methods and attributes an object must possess, without requiring the object's class to explicitly inherit from the protocol. For instance:

```python
from typing import Protocol

class ModuleBuilder(Protocol):
  def __call__(
    self,
    config: TransformerConfig,
    hidden_size: int,
  ) -> nn.Module: ...

@dataclass
class Submodules:
```

```
  linear_qkv: ModuleBuilder
  # ...

class MyModule(MegatronModule):
  def __init__(
    self,
    config: TransformerConfig,
    submodules: Submodules,
  ):
    self.linear_qkv = submodules.linear_qkv(config, config.hidden_size)
```

To put it another way, `Protocol` allows you to define a type hint for an **arbitrary 'usage pattern'**, and the type checker can verify that users provide an object which can be correctly used according to that interface.

Note: `Protocol` provides a way to create extremely powerful *static* type hints, but is *not* intended for use dynamically at runtime. By default, it is not legal to check if an object satisfies a `Protocol` at runtime *at all*. While it is possible to enable runtime checking of a `Protocol`, it is both slow and significantly less featureful (e.g. it does not examine method signatures, only the presence of a method with the same name).

ParamSpec

The `ParamSpec` type (introduced in PEP 612) is used for precisely typing **higher-order functions**: functions that take other functions as arguments or return functions. Specifically, it allows a type checker to correctly relay the argument types of one callable to another, allowing for decorators or wrappers to preserve type hints.

Example:

Python

```python
from typing import Callable, TypeVar, ParamSpec

R = TypeVar('R') # Return type
P = ParamSpec('P') # Parameter specification

def log_call(func: Callable[P, R]) -> Callable[P, R]:
    def wrapper(*args: P.args, **kwargs: P.kwargs) -> R:
        print(f"Calling {func.__name__} with args={args}, kwargs={kwargs}")
        return func(*args, **kwargs)
```

```
    return wrapper

@log_call
def example_function(a: int, b: str, *, debug: bool = False) -> float:
    # ... function logic ...
    return 3.14
```

Megatron at ReflectionAI

ReflectionAI has been using a local fork of Megatron for experimentation since May 2025; we are now migrating to the public fork, and hoping to upstream the improvements we made to our local fork as part of the process. Reflection internally has strict type-hinting requirements which required widespread cleanups of our internal Megatron implementation, and in particular unraveling `ModuleSpec` was necessary in order to avoid `# type: ignore` almost everywhere.

Our use-case was somewhat simplified (since, for instance, we always knew every type that could be provided for each submodule, so could simply use a `Union` of those types), but the provided design is adapted directly from the cleanups we already did internally. These changes allowed us to fully type almost all of Megatron, with only occasional places where we must exempt blocks of code from strict type checking. In particular, this setup provides the following properties:

- It can be done incrementally (so e.g. it can easily coexist with `ModuleSpec` and does not require changes to existing configurations, although existing `ModuleSpec` uses must be converted in order to receive improved type-checking).
- All converted configurations are fully type-checked: when a submodule type is provided, an error will be shown if:
    - The outer module will not pass the correct parameters to initialize it; or
    - The outer module will attempt to provide incorrect arguments to the Module itself when using it; or
    - Any parameters are provided in the configuration which the submodule does not expect.
- All module code is fully type-checked: when a module is constructed, the signature of its `forward` method is completely preserved, so the type checker will enforce that the correct arguments are passed, and will correctly infer the types of the returned values.

Design

The key idea is to define two `Protocol`s to document the expected interface of each submodule.

Builder Protocol

Each module should document how it expects to *instantiate* its submodule, by providing an explicit interface:

```python
from typing import Protocol

class LinearQKVBuilder(Protocol):
  def __call__(
    self,
    input_size: int,
    output_size: int,
    /,
    *,
    config: TransformerConfig,
    # ...
  ) -> '<see next section>': ...

@dataclass
class SelfAttentionSubmodules:
  linear_qkv: LinearQKVBuilder
  # ...

class SelfAttention(MegatronModule):
  def __init__(
    self,
    config: TransformerConfig,
    submodules: SelfAttentionSubmodules,
  ):
    self.linear_qkv = submodules.linear_qkv(
      config.hidden_size,
      self.query_projection_size + 2 * self.kv_projection_size,
      config=config,
    )
```

This provides the same level of flexibility as `ModuleSpec`, while providing complete type-checking of all arguments. Callers can trivially “opt out” of the type-checking by

providing a callable which has no type hinting (or by using `# type: ignore`, of course), but can benefit from type-checking when relevant as well.

One important benefit of this approach is that it makes it much easier to understand how to use standard tools to *adapt* modules so that they can be used correctly, since the interface is provided explicitly instead of implicitly. For instance:

1. If a module definition perfectly matches the interface already, then its type matches the `Protocol` so it can be provided directly:

```python
return ModuleSpec(
  module=SelfAttention,
  params={'attn_mask_type': AttnMaskType.causal},
  submodules=SelfAttentionSubmodules(
    linear_qkv=ColumnParallelNormalLinear,
  ),
)
```

2. If a module type expects additional arguments which won't be provided according to the interface, then `functools.partial` can forward them (and type-checkers fully support it):

```python
return ModuleSpec(
  module=SelfAttention,
  params={'attn_mask_type': AttnMaskType.causal},
  submodules=SelfAttentionSubmodules(
    linear_qkv=functools.partial(MyCustomLinear, debug_mode=True),
  ),
)
```

3. If a module type expects arguments with different names, then a custom adapter function can be used, allowing for easy (and type-checker approved!) incremental name migrations

```python
def custom_linear(
  input_size: int,
  output_size: int,
```

```
  pg_collection: ProcessGroupCollection,
):
  return CustomLinear(
    input_size,
    output_size,
    # TODO: rename to pg_collection
    model_comm_pgs=pg_collection,
  )
return ModuleSpec(
  module=SelfAttention,
  params={'attn_mask_type': AttnMaskType.causal},
  submodules=SelfAttentionSubmodules(
    linear_qkv=custom_linear,
  ),
)
```

This can be done by the caller, or during wide-scale refactorings can be provided by the modules themselves to give callers an easier migration:

Python

```
from typing import Protocol

class SubmoduleBuilder(Protocol):
  def __call__(
    self, pg_collection: ProcessGroupCollection
  ) -> '<see next section>': ...

class OldBuilder(Protocol):
  def __call__(
    self, tp_group: torch.dist.ProcessGroup
  ) -> '<see next section>': ...

class TensorParallelGroupAdapter:
  def __init__(self, inner_builder: OldBuilder):
    self.inner_builder = inner_builder

  def __call__(
    self, pg_collection: ProcessGroupCollection
  ) -> '<see next section>':
    return self.inner_builder(tp_group=pg_collection.tp)

# In configurations:
return ModuleSpec(
```

```
  module=SelfAttention,
  params={'attn_mask_type': AttnMaskType.causal},
  submodules=SelfAttentionSubmodules(
    linear_qkv=TensorParallelGroupAdapter(ColumnParallelNormalLinear),
  ),
)
```

To zoom in on an important design element, note that the interface should explicitly distinguish between *positional arguments* and *keyword arguments*:

```python
class LinearQKVBuilder(Protocol):
  def __call__(
    self,
    input_size: int,
    output_size: int,
    # Arguments before / are positional-only.
    /,
    *,
    # Arguments after * are keyword-only.
    config: TransformerConfig,
  )

# below, when using:
  self.linear_qkv = submodules.linear_qkv(
      # First two arguments are provided based on position:
      config.hidden_size,
      self.query_projection_size + 2 * self.kv_projection_size,
      # Other arguments are provided by name:
      config=config,
    )
```

This provides the maximum amount of flexibility to the modules which must match this `Protocol`: the type-checker will not enforce the names of arguments that are positional-only, and will not enforce the order of parameters which are keyword-only, ensuring that submodule implementations are not unduly restricted by needing to match the `Protocol`.

Module Protocol

A second `Protocol` should be defined to describe the way in which the submodule will actually be *used*. In trying to do this, naive approaches run into a bit of a problem: `torch.nn.Module` *itself* is poorly typed! The correct usage of a `torch.nn.Module` instance is defined by the signature of its `forward` method, but users are intended to actually use `__call__` instead, which has an opaque/untyped signature. This presents us with a bit of a catch-22:

- If we created a `Protocol` which only checked for a particular signature of `__call__`, then configurations would not be type-checked (since `Module.__call__` is untyped and therefore would always match).
- If we created a `Protocol` which only checked for a particular signature of `forward`, then configurations would be type checked, but the *usage* of the module would not (since it would be using `__call__` instead, which would still be untyped).
- If we created a `Protocol` which checked for *both* signatures, then this would technically work, but would double the boilerplate needed, and there would be nothing actually enforcing that the two signatures remained the same (potentially leading to extremely confusing bugs).

But we can leverage a nice combination of `ParamSpec` and `Protocol` to propagate type hints from `forward` when actually using the `Module`. We can introduce the following helper method (Reflection has it in a library called `typed_torch`) which massively improves the type-checking for modules and avoids this problem[6]:

Python

```python
# typed_torch.py
class _Module[**P, R](Protocol):
  """Protocol allowing us to unwrap `forward`."""

  def forward(self: Self, *args: P.args, **kwargs: P.kwargs) -> R: ...

def apply_module[**P, R](m: _Module[P, R]) -> Callable[P, R]:
  """Returns a typed function that applies the given module.

  Args:
    m: An instance of a subclass of `torch.nn.Module` to apply.

  Returns:
```

[6] It is presented exactly how Reflection has it defined internally, which is aimed at python3.12; it can be adapted to support 3.10 without much work.

```
    That module unchanged, but with correct type hints.
  """
  assert issubclass(type(m), torch.nn.Module), (
    f'{type(m)} is not a subclass of torch.nn.Module'
  )
  return m  # type: ignore
```

Even outside the `ModuleSpec` replacement, Reflection has found this tactic fairly useful for typing code within `Module`s in a fairly painless way (simply letting callers use `typed_torch.apply_module(module)(params)` instead of `module(params)` to enable type-checking). But it also provides an effective means of tying together a `Protocol` which actually enforces that a `Module` have the correct signature, and the type-checking of the *use* of that module, since it provides a natural way of "unwrapping" the `Protocol` when needed:

Python

```python
class LinearQKV(Protocol):
  def forward(
    self, hidden_states: torch.Tensor, /
  ) -> Tuple[torch.Tensor, Optional[torch.Tensor]]: ...

  def backward_dw() -> None: ...

class LinearQKVBuilder(Protocol):
  def __call__(self) -> LinearQKV: ...

class SelfAttention(MegatronModule):
  def forward(
    hidden_states: torch.Tensor,
    # ...
  ):
    mixed_qkv, _ = typed_torch.apply_module(self.linear_qkv)(hidden_states)
    # ...

  def backward_dw() -> None:
    self.linear_qkv.backward_dw()
```

By defining a `Protocol` which encodes the signature expected for `forward`, using this as the return type of the 'Builder', and using `typed_torch.apply_module` where we would

normally just call the module directly, we ensure that the type hints of the module provided at configuration time are *completely* propagated and enforced from beginning to end.

Migration Plan

Draft PR showing Step 1 and migrating a few ModuleSpec uses: https://github.com/NVIDIA/Megatron-LM/pull/2454

Step 1: Preparation

Make ModuleSpec Callable

To provide backward compatibility with existing `ModuleSpec` usages, we need to provide a clean way to migrate from `build_module`; since `Builder` protocols will simply be callable directly, we can easily do this by making `ModuleSpec` technically satisfy `Builder` protocols (albeit without the type-safety aspect):

```python
@dataclass
class ModuleSpec:
    module: Union[Tuple, type]
    params: dict = field(default_factory=lambda: {})
    submodules: type = None

    def __call__(self, *args: Any, **kwargs: Any) -> Any:
        return build_module(self, *args, **kwargs)
```

With this in place, `ModuleSpec` can continue to be used wherever we add a `Builder` interface, and the relevant callsite can be migrated from `build_module(submodules.x, ...)` to `submodules.x(...)` without changing the behavior.[7]

Submit typed_torch

We need `typed_torch` (or its equivalent) available within Megatron to support unwrapping these `Protocol`s.

[7] This *will* drop support for the `types.FunctionType` special-casing in `build_module`; however, this is probably most safely supported with an adapter that wraps a function in a Module, rather than continuing to support functions directly.

Step 2: Large-Scale Migration of Existing ModuleSpecs

Individual modules can be converted in place in effectively any order. In our experience, most cases are very easy to refactor; however, certain modules which are doing confusing things (like customizing `kwargs` before building the module based on properties of the `ModuleSpec`, or directly examining properties of submodules in order to change their own configuration) can require more care in order to clean things up correctly.

- In every case we tried, migrating to a `Protocol`-based approach (e.g. by using an Adapter pattern as shown above, or restructuring so that configuration was passed directly to outer modules instead of extracted from submodules) led to more boilerplate but significantly easier-to-debug code.
- Configurations do not need to be updated at the same time as the underlying modules (thanks to Step 1), but can be updated immediately after.
- Whenever a submodule is updated to be `Protocol`-based, the corresponding module code will need to simultaneously swap to (1) stop using `build_module`, and (2) start using `apply_module`.
- There are a number of cases where existing `Submodules` objects are reused by different modules, but in reality those modules do not have the same interface. This means that those modules will need to 'fork' their `Submodules` objects to let their underlying protocols diverge (and this is probably a good thing).

Step 3: Optional Cleanups

Once the Large-Scale Migration is complete, there are several nice-to-haves we can follow up with:

- Delete `ModuleSpec` and friends, since they are no longer needed. (This is completely optional, and can be delayed indefinitely to support external users who continue to need it. In practice, though, `ModuleSpec` is just `functools.partial` in disguise so is pretty much trivial to refactor away.)
- Provide stubs or wrapper libraries to make type-checkers understand optional imports like `transformer_engine` or `kitchen`. For instance, `BackendSpecProvider` can have its interface redefined based on the relevant `Protocol`s needed (rather than simply returning types directly), so that implementations can do type-checking of their own underlying types.
- Enable a type-checker in CI/CD to *enforce* the correctness of type hints, rather than just locally being able to see them in an IDE. This would require a larger-scale

cleanup, such as by adding `# type: ignore` comments throughout and selectively removing them as type hints are added.[8]

Known Limitations

- The effectiveness of type-checking will still be limited by the presence and correctness of type hints in the modules being used. In order to take full advantage of this design, there must be an appetite for providing effective type hints to the modules being defined within Megatron.
- This `Protocol`-based design introduces a non-trivial amount of boilerplate due to the number of arguments that most callsites have. While this is annoying, the explicitness does make the code more self-documenting, and the fact that the interface is then continuously checked for correctness by the type-checker means that it ends up having a significantly lower maintenance cost than alternatives.
- Several of the features being used (`Protocol`, `ParamSpec`, `typed_torch`) are advanced and may not be as well-understood by users. However, this is largely offset by two things:
    - The users don't need to directly *use* any of the features, they simply need to construct configurations which follow their example (and a type checker can easily confirm for them that they've done so correctly).
    - Unlike `ModuleSpec` (which is both complex *and* unique to Megatron), `Protocol` and `ParamSpec` have large amounts of public documentation and examples of their use. The helper in `typed_torch` would be new to readers, but a quick look at the docstring would make it quickly apparent that it has no effect on anything except typing, so can basically be ignored.
- Lazy importation of module types by name is no longer a built-in feature (partially since this feature is a nightmare from the perspective of both type checkers and general understandability). However, this is still simple for configurers to manually do if they want - they can simply fulfill the Builder `Protocol` with a method that lazily imports the appropriate class and then constructs an instance of it. As a bonus, type-checkers will actually be able to understand this pattern too (as long as the caller uses a normal import statement rather than using `importlib`).

[8] This is how Reflection approached this internally - note that we also enabled a setting in Pyright which forces `# type: ignore` comments to be *removed* on lines where they are unneeded, which ensured that the codebase monotonically improved as types were made available.