Software Library for Generative Model Applications

by

Carlos Hernandez

B.S. in Aerospace Engineering and in Electrical Engineering and Computer Science, Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by: Carlos Hernandez
Department of Electrical Engineering and Computer Science
May 19, 2023

Certified by: Aude Oliva
MIT director of the MIT-IBM Watson AI Lab
Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Software Library for Generative Model Applications

by

Carlos Hernandez

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The generation of data by machine learning models is a powerful concept that has impacted the field of Artificial Intelligence in the past few years. In this thesis, we focus on building a software library to facilitate the workflow, evaluation, and analysis of generative models. Our work is primarily aimed at helping a specialty chemicals company use a state of the art molecule generation model for their specific applications. We reference the body of work containing the model as DEG, short for Data-Efficient Graph Grammar Learning for Molecular Generation [16]. DEG is capable of creating synthesizable molecules from small amounts of data, making it quite attractive for companies looking for practical methods to explore new molecules. As an overarching goal, we will design our library to incorporate other types of generative models and become a tool that the field can benefit from.

Thesis Supervisor: Aude Oliva
Title: MIT director of the MIT–IBM Watson AI Lab

Acknowledgments

I would like to thank the MIT/IBM Watson AI Lab for all of the support and guidance provided to me. I would especially like to thank my mentor, John Henning, and my teammate, Aditya Prasad, who supported me with everything throughout my time in the lab and helped me become a better engineer. In addition, I would like to thank my manager, Lee Martie, for trusting me with a role in his team and advising me during the project.

Contents

1 Thesis **13**

1.1 Introduction 13

1.1.1 Generative Models 14

1.1.2 Molecule Generation 14

1.1.3 Formal Grammar 16

1.1.4 Graph Grammar 16

1.2 Related Work 17

1.2.1 GT4SD 17

1.2.2 TorchGAN 18

1.3 Customer Requirements 18

1.4 Original Codebase 19

1.5 Initial Ideas 23

1.5.1 Modules 23

1.5.2 Abstractions 24

1.5.3 Limitations 25

1.6 Design and Implementation 26

1.6.1 Package Diagram 27

1.6.2 API 27

1.6.3 Jobs 31

1.6.4 Specs 35

1.6.5 Metrics 43

1.6.6 Data Logging 47

1.6.7 Jobify 50
1.6.8 Engineering Choices 51
1.6.9 Remote Version 53
1.7 Future Work 54
1.8 Generalizing to Other Domains 54
1.9 Conclusion 55

List of Figures

1-1 DEG Training Pipeline: the middle image shows how rules are extracted from the training data and how they map to molecule subgraphs. These rules are then used to generate new molecules, shown on the right-most image 17
1-2 DEG Training Sequence Diagram 20
1-3 DEG Package Diagram 22
1-4 DEG Potential Abstractions 25
1-5 DEG2.0 Package Diagram 27
1-6 REST API Structure 28
1-7 DEG2.0 Local 30
1-8 DEG2.0 Remote 31
1-9 Train Spec 38
1-10 Evaluate Spec 38
1-11 Generate Spec 38
1-12 IOManager Class 47
1-13 All Artifacts Across Training 49
1-14 Best Artifacts Across Training 49
1-15 No Artifacts 49
1-16 File Structure 49

List of Tables

1.1 Time to run Retro Star across different sample amounts using 5 processes 53

Chapter 1

Thesis

1.1 Introduction

A general issue with AI applications is that adapting research-tailored codebases for a new domain or application is often difficult. The code is normally designed in the context of the research, and the API's are made to handle situations within that context.

This issue is readily visible in the case of the chemicals company we work with. They would like to use the algorithms developed in DEG to create graph grammars that can generate synthesizable molecules. However, they do not want to be restricted by the limitations of the research codebase and have asked us to extend it to allow for different use cases. For example, one desired extension they would like to have is the ability to optimize the learned grammars with different objective functions, or metrics, as they are called in DEG. Theoretically, they could go into the code and do this themselves, but they would need to read line-by-line to see where to make the change and ensure the new code does not break any of the existing logic. Figuring this out would then require an understanding of how the entire codebase works. And if the code does not have proper abstractions and structure to begin with, a future change might require modifying what they have already changed before, creating a maintainability problem. In general, software should be designed such that its users can use the features that interest them without needing to delve into the code nor

modify it [3]. It should also be implemented with scalability in mind, ensuring that future additions will not break the current version nor require major refactoring [6].

In this thesis, we dive into the motivation, design, and implementation of DEG2.0, a software library made to facilitate the use of DEG for our customer (the chemicals company). We also frame the discussion within the context of a secondary goal, generalizing the library to work with other types of generative models, focusing initially on expanding DEG to different domains.

We begin by introducing some of the background needed to appreciate the importance of generative models and establish the context of the specific problem we tackled.

1.1.1 Generative Models

The goal of generative models is to compress the underlying distribution of data such as natural images, audio waveforms, or text, into a set of parameters that can be used to generate new data. They can take many forms, such as Generative Adversarial Networks (GAN) [15], Variational Autoencoders (VAE) [21], and Autoregressive Models [25]. These models have important applications in real-world problems such as denoising images and generating synthetic data. They have even gained popularity within the art community through the unique images generated by models like Google's BigGAN [4] and OpenAI's DALL-E [29].

Given the potential generative models have to attack a variety of problems, designing flexible software to leverage their implementations is an important task.

1.1.2 Molecule Generation

When it comes to designing molecules to meet a specific goal, chemists often rely on their intuition of what types of molecules could be useful for their given problem [23]. Having a computer compactly learn the distribution of useful molecules and sample from it would allow chemists to focus on synthesizing and testing molecules rather than designing them from scratch. Generative models are a natural solution for this.

There are many aspects that make this problem difficult and interesting, making it an important research topic within the field.

Most approaches for generating molecules are based on deep neural networks. Deep learning is a data hungry approach and requires large datasets for training (in the order of 10s of thousands). The issue with this is that in practice, acquiring chemical data is too difficult to build datasets of that magnitude [36]. Even if the data was available, there is no guarantee that the generated molecules are synthesizable. Another possible approach is to generate molecules from a "formal grammar". A formal grammar can be thought of as a set of rules from which to build a language. In our case, this language is a language of molecules (details in the following section). With grammar approaches, one can make some assumptions about the ouputs since they are constrained by the grammar rules. Also, less data is required to reach good performance since the approach moves away from deep learning and instead uses a smaller, grammar-based model. Unfortunately, an issue with this approach is that creating the grammar can be difficult. Some experts have tried to manually specify it, but this is not scalable since the grammar would need to be re-created each time the training data changes and humans are prone to making mistakes [8].

DEG deals with all of these issues and adds additional contributions. By learning a graph grammar from the training data, the model generates molecules that are automatically in the training set distribution. By using a grammar approach, it reaches competitive performance with significantly less data (in the order of 10s of samples). Also, compared to its most similar counterpart [20], it does better at capturing distribution-specific properties (e.g. molecular class substructures) by learning sub-graphs rather than individual atoms. A key contribution from DEG is the ability to optimize a grammar with respect to provided metrics. For example, by optimizing the grammar with respect to the distance between the training set and the generated set, DEG can generate molecules that are different from the training set. These metrics allow users of DEG to optimize grammars with respect to different objective functions, which is one of the key features our customer is interested in.

For more background on formal grammars and graph grammars in the context of

DEG, read the following two sections.

1.1.3 Formal Grammar

Similar to how the English language has a specific set of rules for combining words into sentences, rules can be defined for generating a subset of all molecules. These rules form the grammar that DEG tries to learn. Below, we describe a formal grammar in the case of strings.

A formal grammar $G = (N, \Sigma, P, X)$ has a finite set of non-terminal symbols N, a finite set of terminal symbols Σ, a set of production rules P, and an initial symbol X. It describes how to build strings from a language's alphabet starting from X and using rules $p_i \in P$ where $p_i : LHS \rightarrow RHS$ (left-hand side to right-hand side). When a non-terminal symbol matches the LHS of a rule p_i, then the symbol is replaced with p_i's RHS. This process keeps going until there are no more non-terminal symbols left in the string.

1.1.4 Graph Grammar

In the case of molecule generation, this grammar is a graph grammar. The concept is the same, except the grammar involves molecule graphs instead of strings. These graphs follow naturally from real representations of molecules, with each atom being a node and bonds being hyperedges. Both sides of each rule p_i are a subgraph of the molecule graph. In order to determine which rule to apply, subgraph matching [14] is used to check whether the current graph has a component equivalent to the rule's LHS.

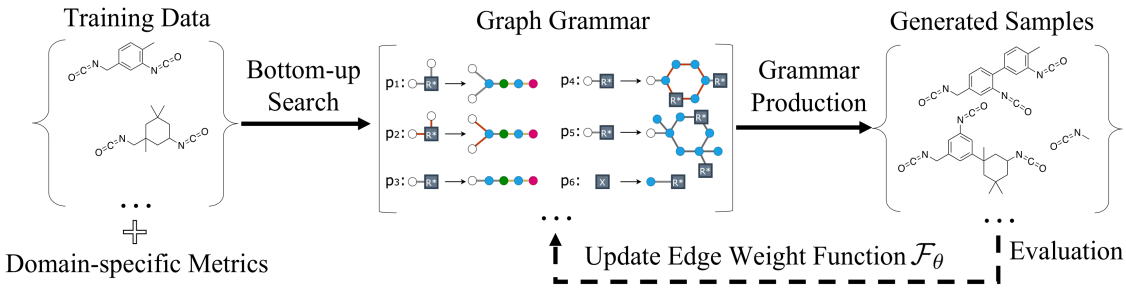

Figure 1-1: DEG Training Pipeline: the middle image shows how rules are extracted from the training data and how they map to molecule sub-graphs. These rules are then used to generate new molecules, shown on the right-most image

1.2 Related Work

In this section, we will highlight previous work done in developing software libraries for generative model applications.

1.2.1 GT4SD

GT4SD, short for Generative Toolkit for Scientific Discovery, is an open-source platform for making generative models easier to use [37]. It is written in Python, the same language as DEG and DEG2.0, and can be used through a command-line API. GT4SD's main responsibilities are training and evaluating models and generating data from them. It has a list of supported models one can quickly play with. Our approach differs from GT4SD in two main areas: functionality and supported models. Our immediate use case with the chemicals company will require making a library that can work with different datasets and a varying experiment workflow, leading to a different library that can do more than just training and evaluating models. Also, given that we must support DEG's model, our architecture is tailored for the use of DEG and may require some changes to support the variety of models found in GT4SD in the future.

1.2.2 TorchGAN

TorchGAN is a framework for simple and efficient training of GANs based on PyTorch [26]. PyTorch is one of the top machine learning frameworks, so it provides great inspiration for us to build from [27]. Some differences between TorchGAN and DEG2.0 is that TorchGAN is specifically made for General Adversarial Networks, focusing on using existing models and designing new ones. With our library, we aim to support more types of generative models, focusing on grammars first. Also, by placing less emphasis on the design of new models, we may be able to provide better support/functionality for existing ones.

1.3 Customer Requirements

To get the project started, our customer sent us a list of requirements outlining what they were looking for. These were not necessarily hard requirements, but they provided some context and served as guidance throughout the project. Ultimately, they wanted us to make a library that would facilitate their experiment workflow.

Their initial requirements can be summarized as two main points:

1. Turn DEG into an easy-to-use Python library that can set up and run different experiments quickly and scalably.

2. Create an architecture that enables DEG to run remotely within containers using a standardized API.

These points are naturally general and open-ended, leading to many discussions within the team about the reasons why the original implementation of DEG was difficult to use and about the ways we could 'standardize' the API while keeping it consistent between local and remote workflows. Guided by these requirements, software engineering best practices, and our knowledge of what AI applications should look like, we got started.

1.4 Original Codebase

Before beginning to change anything, we took the time to understand the original implementation of DEG.

To use DEG, one can clone `https://github.com/gmh14/data_efficient_grammar` into one's local machine and follow the instructions in the `README`. The `README` walks through the initial setup and installation, and asks to activate the main Conda environment [2].

```
conda activate DEG
```

Once the environment is activated, one can run DEG with the following line:

```
python main.py --training_data=./datasets/**dataset_path**
```

Running this command in a terminal is the only predefined way to interact with the code. The command runs the `main.py` file with the `training_data` flag specifying the training dataset path. This is one of many flags that can be used to change the default parameters that DEG runs with. Other flag examples are `max_epochs` and `learning_rate` which set parameters for the learning algorithm. One can see all the possible flags within `main.py` between lines 178-191 .

The entry-point of `main.py` takes care of parsing the flags given in the command and runs a function called `learn()`. This function is the main workhorse of DEG. It learns a grammar from a training dataset by iteratively generating and evaluating molecules, optimizing the production rules that make the grammar. The details of this function can be seen visually in figure 1-2.

The blocks at the top of figure 1-2 show the different Python files that are used within the function. On the top left we can see the block for `main.py` and its connection to `learn()`. One could read through the diagram step-by-step to get a full understanding of what the function is doing, but for now, I will highlight only the important pieces. We can see that each piece of the function is labeled with an ID number. Near the bottom left, we see IDs 20 and 21 labeling the workflows `evaluate_grammar` and

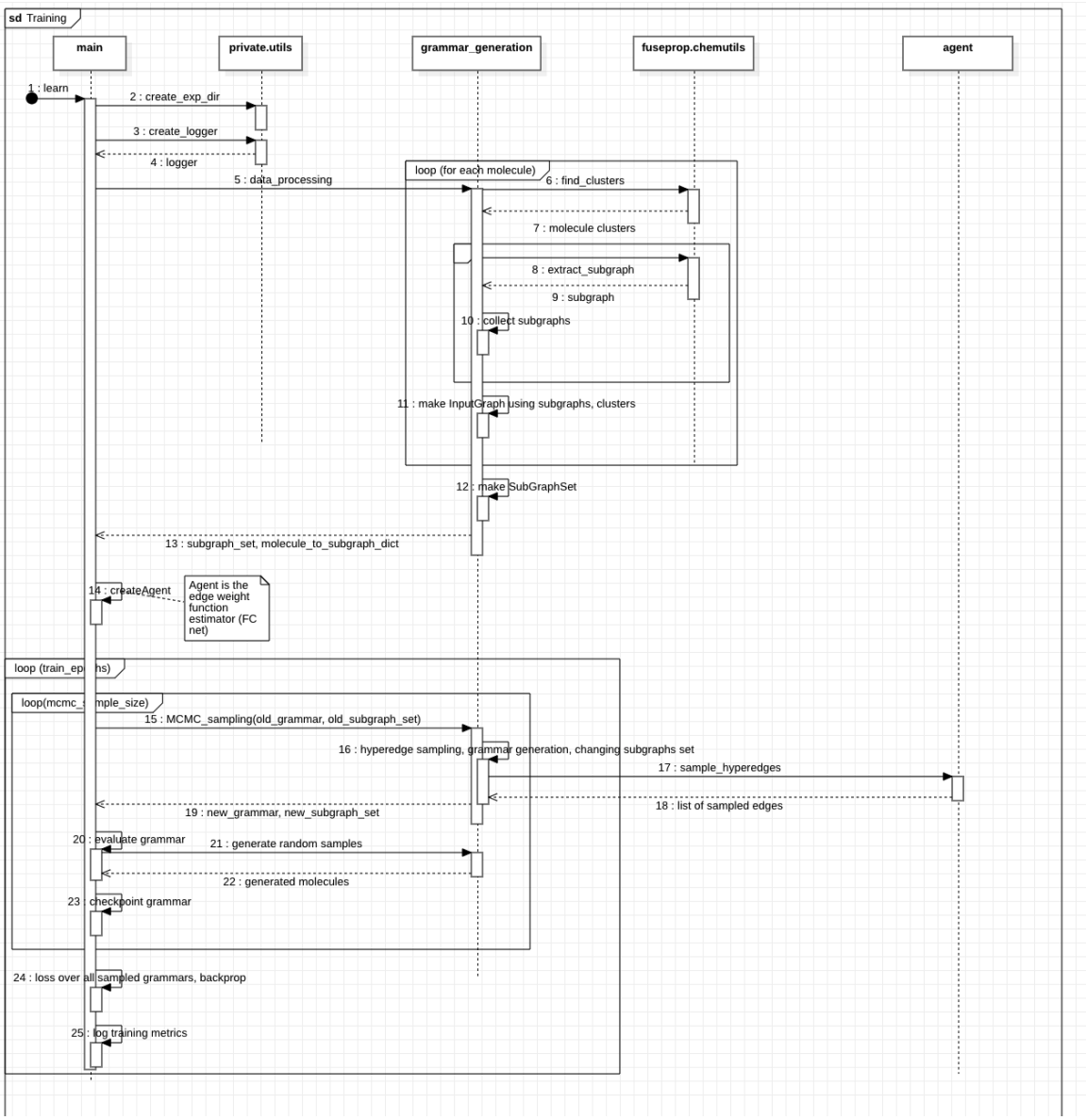

Figure 1-2: DEG Training Sequence Diagram

generate_random_samples. I say "workflows" rather than functions because the diagram is high-level, so the naming does not necessarily correspond to real functions in the code. The evaluate_grammar workflow takes in a new_grammar and uses molecules from generate_random_samples to compute an evaluation of how good the grammar is in that current iteration. This rating is computed with respect to hard-coded metrics which get activated through stacked if/else statements checking which metrics are currently in use. Based on this evaluation, the algorithm computes a loss and tries to minimize it.

The rest of the blocks are helpers for learn(). grammar_generaton.py contains tools for making a grammar, such as processing the training data and making a graph from it. private.utils has, among other files, the graph convolutional network used for feature extraction (GCN) and a file named grammar.py where the code for the ProductionRule object lies. fuseprop.chemutils deals with the chemistry side of things such as converting SMILES (Simplified Molecular Input Line Entry System) strings to molecules and vice versa. agent holds the function estimator for the edge weights in the production rule graph.

These are the files and objects that are directly imported into main.py, but there is more code in DEG outside of main.py that also helps out in the learning process. An important example of this is the retro_star package. retro_star is a package that takes in generated molecules and decides how synthesizable they are. It is used for computing the evaluation of the Retro Star metric. In DEG, one has to run this package in a separate terminal as a separate process:

```
conda activate retro_star_env
bash retro_star_listener.sh **num_processes**
```

The bash script retro_star_listener.sh launches a num_processes amount of retro_star_listener.py's. Each one takes about 5Gb of memory since the entire model is copied each time. This is an inefficiency that caps the number of processes one can run depending on the machine being used. This script reads from a file that is written to by learn() in main.py. When running the Retro Star metric during evaluation, learn() will write generated molecules as strings to a "sender file" and

`retro_star` will read these molecule strings, convert them to molecule objects, compute how synthesizable they are, and write these evaluations back to a "receiver file" which `learn()` is reading. This enables parallelization in the Retro Star evaluation process which speeds up the learning algorithm. The more processes, the faster these evaluations run.

All of these pieces fit together to make the DEG codebase. In Figure 1-3, we can see a visual representation of everything mentioned above. Arrows display the dependencies among different units of codebase. If an arrow points from block A to block B, then block A uses block B.

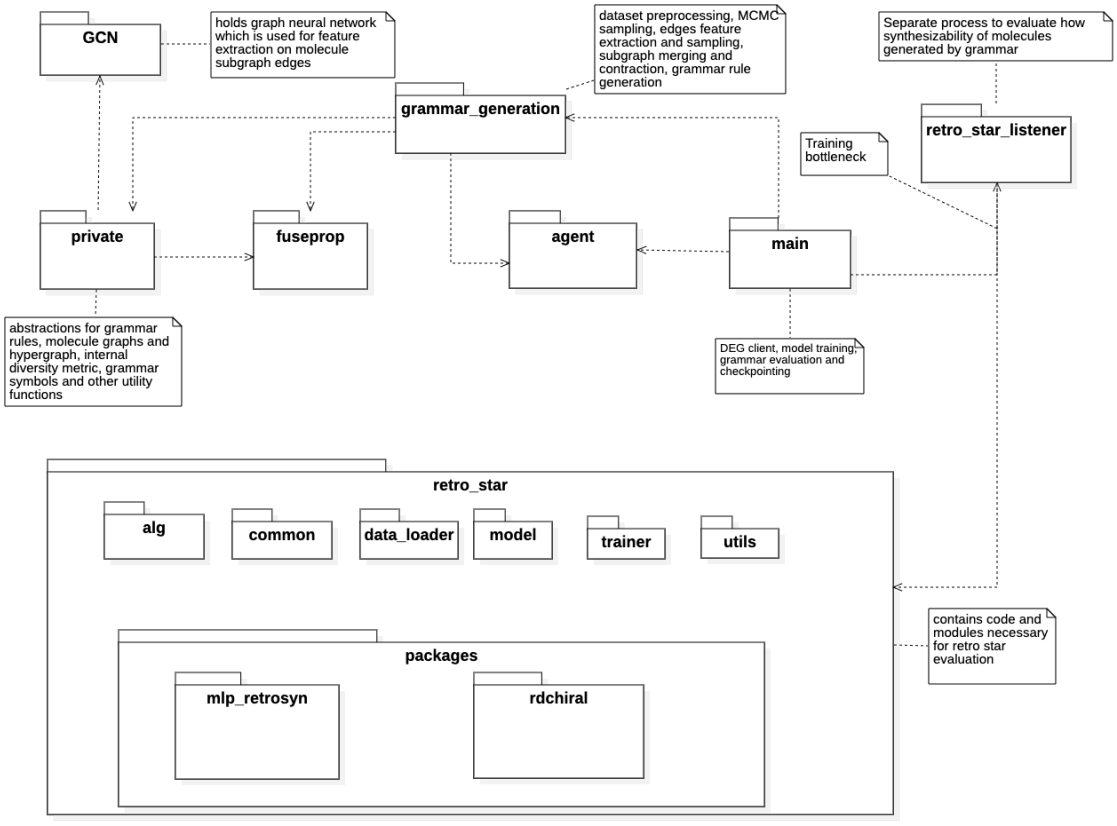

Figure 1-3: DEG Package Diagram

1.5 Initial Ideas

Now that we have an overall idea of how the code works, we can begin talking about the initial thoughts we had during the brainstorming phase of the project. It is worth reiterating that DEG was made for a specific purpose, to perform the science needed to publish a research paper, and it did very well at achieving this. Though the following sections motivate future changes to DEG, these changes are most valuable in the context of a library meant to be more general-purpose, so it makes sense for the authors of DEG to not have considered them. With this being said, we start by exploring DEG with a more critical eye.

1.5.1 Modules

As mentioned in the introduction, we focus on two different goals. The main goal is to make a library that meets our customer's requirements. The secondary goal is to generalize the library and extend it to new domains. Simply meeting all of the customer's requirements can lead to a library that is very specific to the use case they immediately want. This can be dangerous because they do not fully know what they want and are expecting their requirements to change in the future. Due to this, thinking through the design of the library through the lens of the secondary goal is actually beneficial to the main goal.

A good first step for accomplishing this is to try to separate the code into individual modules. We can think of a module as a block of code one can easily plug-and-play into different workflows. For this section, it will be helpful to reference figure 1-2 again. Looking at `learn()`, one can begin to see pieces that could naturally form individual modules. The `evaluate_grammar` and `generate_random_samples` workflows could be their own functions and one could call them without needing to run `learn()`. This would enable the user to evaluate multiple grammars and generate molecules flexibly without running extraneous code. IDs 3, 4, 23, and 25 show the need of having a logging module to keep track of training artifacts such as performance and grammar checkpoints. ID 5 motivates a data processing module, 6 motivates an edge-weight,

function-estimator module, and 15 motivates an MCMC sampling module. Digging a little deeper into `evaluate_grammar`, we see that it makes the evaluation based on some metrics. Redesigning this process such that metrics can be used interchangeably would allow our client to optimize grammars with respect to different objectives. This motivates a metrics module.

In DEG, some of these modules actually live as functions in separate "units" already. For example, the suggested data processing and MCMC modules are implemented as functions (`data_processing()` and `MCMC_sampling()`) in `grammar_gener--ation.py`. Others might simply be `if/else` statements, like the metrics. The actual implementation of the modules is not as important as whether or not they form individual logical units. If they do, we can implement the code blocks as objects or functions to design an API that reaps from flexibility benefits similar to the example given above about the evaluate and generate functions.

1.5.2 Abstractions

Very little from the DEG code is actually abstracted from the user. Other than the command-line call to `main.py`, there is no API for interacting with the code. The only way to get anything to happen other than a call to `learn()` is to move existing code around or write new code. In general, software should be designed with the target user in mind. It should have the proper levels of abstraction to allow users to quickly understand the workflow and use it to complete their responsibilities [32]. The target user should never need to edit the "business logic" code of the library. In the case that they want to write new code, as is the case with our customer when it comes to metrics, the API should provide a template and the library should be robust to mistakes in the new code.

As described above, abstractions play an important role when it comes to determining how users interact with software. Another place where abstractions are important is within the code itself. In Object Oriented Programming, this is done through inheritance [22]. Before implementing the new library, we took the time to make as many code abstractions as possible. This thought process is similar to

modularizing the code, except it is more oriented towards pieces that naturally fit as objects. Thinking through these abstractions was a key step towards designing for the secondary goal of the project. An example of an abstraction that would facilitate transitions to a new domain is a grammar abstraction. When evaluating a grammar, the evaluation function could be very similar for any type of grammar, whether the grammar is for molecules or for a different domain. We could pass a molecule grammar or a circuit grammar into the function as long as both objects are grammars. There are many more potential abstractions that could be used in DEG. Figure 1-4 shows the variety of abstractions we came up with during the brainstorming phase of the project.

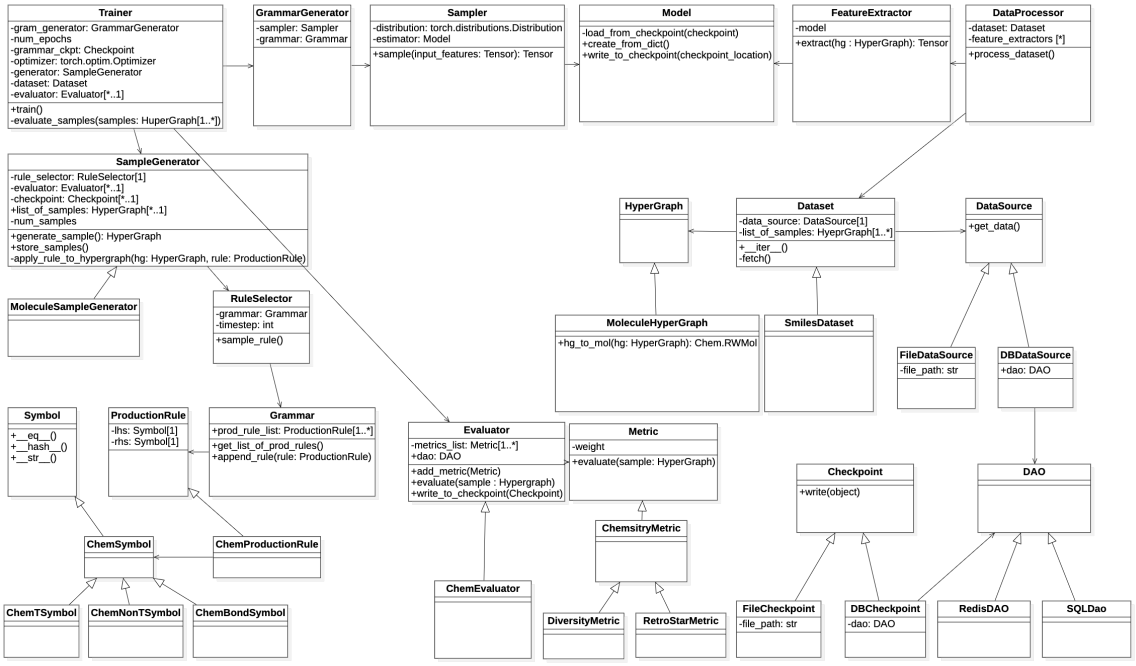

Figure 1-4: DEG Potential Abstractions

1.5.3 Limitations

In the sections above, we introduced two overarching ideas which, at a high level, motivate much of the work done in the project. Before talking about the design and implementation of the new library, we summarize the conversation above into

a concise list highlighting the limitations of DEG. The following section focuses on our solutions to these limitations, so one could use this list as a reference point while reading.

- Design Issues

    1. No general API abstracting business logic from user workflow
    2. No API for doing evaluation or molecule generation using existing grammar
    3. Minimal modularity within code
    4. Minimal flexibility in possible experiments
    5. No scalable way for creating and using metrics
    6. Minimal persistence for tracking experiments
    7. No remote containerized workflow

- Engineering Issues

    1. If using Retro Star metric, user has to run a shell script for evaluation in a separate terminal before `learn()` can be run. This also requires manual cleanup after run finishes
    2. File transfer is used to speed up Retro Star metric
    3. Two separate Conda environment installations needed
    4. Manual installations of Python packages from source

1.6 Design and Implementation

In this section, we dive into the details of DEG2.0. DEG2.0 was designed to work in both local and remote cases, as requested by our customer. Although much of the work done in this project focuses on the local implementation, we will see throughout this section how the decisions we make keep in mind the future transition towards the remote version. We begin by describing DEG2.0 at a high-level, showing the package

diagram and explaining the API, and then go into the details of the main components in the following subsections.

1.6.1 Package Diagram

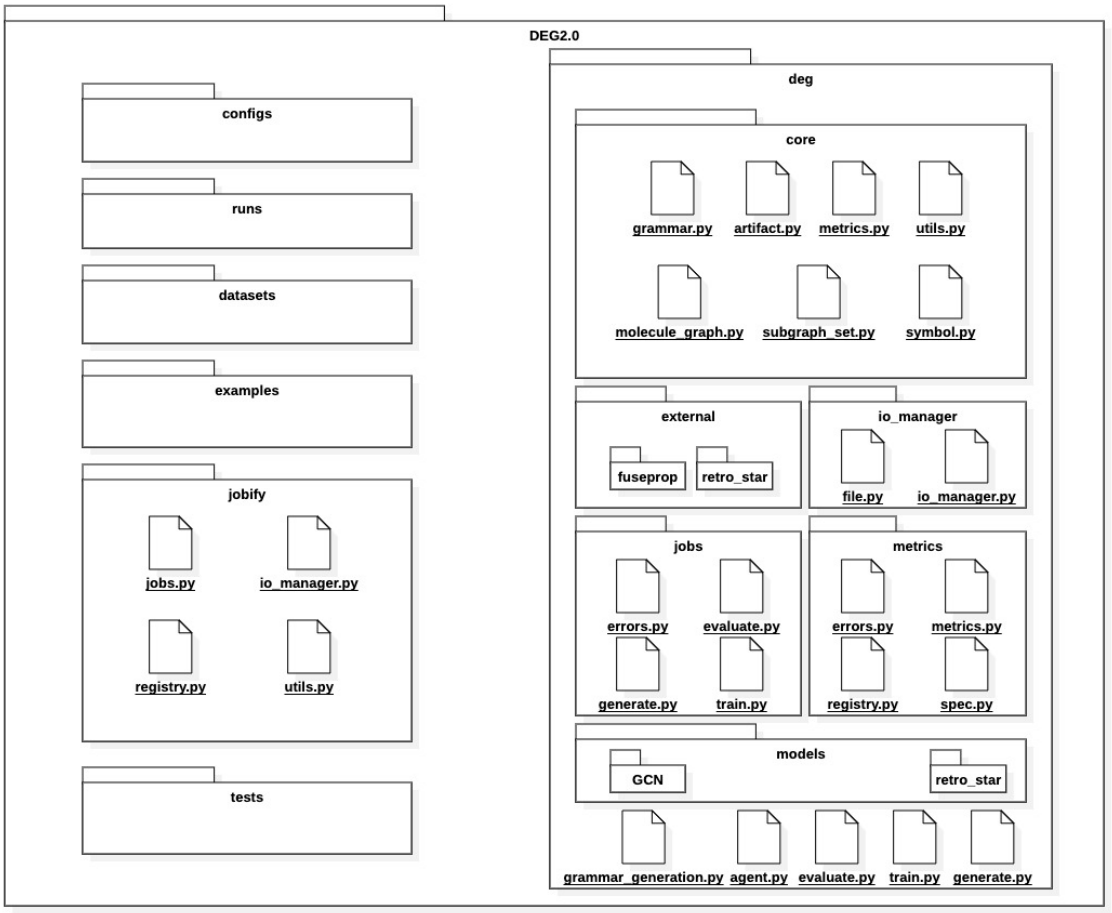

Figure 1-5: DEG2.0 Package Diagram

1.6.2 API

API stands for Application Programming Interface. An API is basically a contract that defines how programs talk to each other [17]. To meet our customer's requirements, we needed the API to function in both local and remote cases. To accomplish this, we built an architecture inspired by REST APIs [24]. REST is an architectural style for providing standards between computer systems on the web. Some key terms

used to describe REST APIs are client, resource, and server. A client is the program using the API. The client makes requests to the API in order to retrieve information or change something within the application. In our case, the client is the program the customer is exposed to. A resource is any piece of information the API can provide the client. A server is a computer or program used by the application that receives client requests. The server contains resources that the client wants. It sends these resources as responses without giving the client direct access to the content stored in its database. Other terms for client and server are "front-end" and "back-end". These are commonly used when talking about full-stack development.

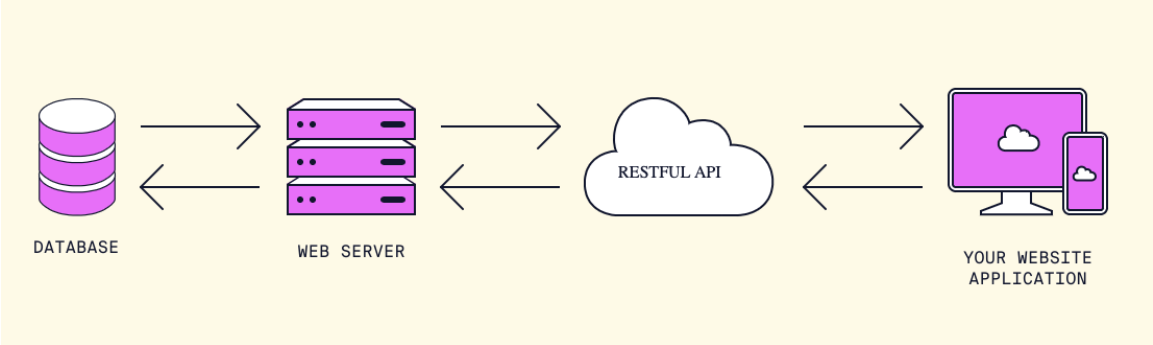

Figure 1-6: REST API Structure

To benefit from the functionality that REST provides, APIs must follow the following six requirements: client-server separation, uniform interface, stateless, layered system, cacheable, and code on demand [19]. We focus mostly on the first four requirements since the last two are less relevant for our application. Client-server separation refers to the ways the client and server interact. Under REST architecture, the client and server can only interact through requests sent by the client and responses sent by the server. All interactions are initiated by the client. Uniform interface means that requests and responses must follow a common protocol. For most REST APIs, this common protocol is HTTP, the Hyper-Text Transfer Protocol used in the internet. Stateless means that every interaction with the API is independent, with each request and response containing all the information necessary to complete the interaction. Finally, layered system means that no matter how the server side is structured,

messages between the client and the target server should always be formatted and processed the same way. This allows server systems to be modified without affecting the core request-response behavior.

These guidelines provide a good model for the workflow our customer wants. Their first requirement calls for an easy-to-use, scalable library. Following client-server separation allows us to make a client that can flexibly run experiments without needing to change the business logic. As we saw in section 1.4, DEG only has one workflow which is parameterized with some command-line flags. Running an additional workflow, such as evaluating molecules, would require adding code to DEG's client side (the `main()` function in `main.py`) and integrating the code to the server side (the rest of DEG). Instead, we focus on building a client that allows for all the possible workflows the user may need so that they never have to worry about the server side. Also, this separation gives us the liberty to abstract the client program as much as we want depending on how high-level we want the user's workflow to be. The third guideline, keeping the system stateless, ensures DEG2.0 is predictable and understandable for the user since the actions done in the server are determined by the information passed in the current request. The second and fourth guidelines, uniform interface and robustness to layering, ensure the library is scalable since future changes to either the client or the server should not break the system as long as the guidelines are met.

By following the REST API structure, we were able to decouple the design and implementation of the library from the compute resource used to run it. Note that in local implementations of REST APIs, the client and server live on the same machine and are only separated structurally. This means that the requests made by the client do not need to travel through the internet and can instead simply run on the same machine. So the local version does not need to follow REST guidelines to function. However, making the local API RESTful facilitates the transition to the remote version. All we would need to do to make a remote version is route the requests to remote endpoints where the code runs. Designing the API following REST enabled us to focus mostly on the local version for the first release of DEG2.0 knowing we would be able to use a lot of the same code for the remote version.

The overall information flow we designed is the following. In the client, the user builds a "spec" defining the "job" he or she would like to run. The client sends this spec to the server where it is parsed through a package called `jobify`. Jobify runs a spec checker ensuring the client inputted everything correctly, and if so, creates a job and runs it. In the local version, this job runs on the same machine as the client (the user's computer for example). In the remote version, the job runs in a container. In both versions, the specs can be defined in a Jupyter notebook or in files. This structure can be seen in figures 1-7 and 1-8. In the following sections, we go through the details of these individual pieces, as well as important design and implementation choices made to allow this to work.

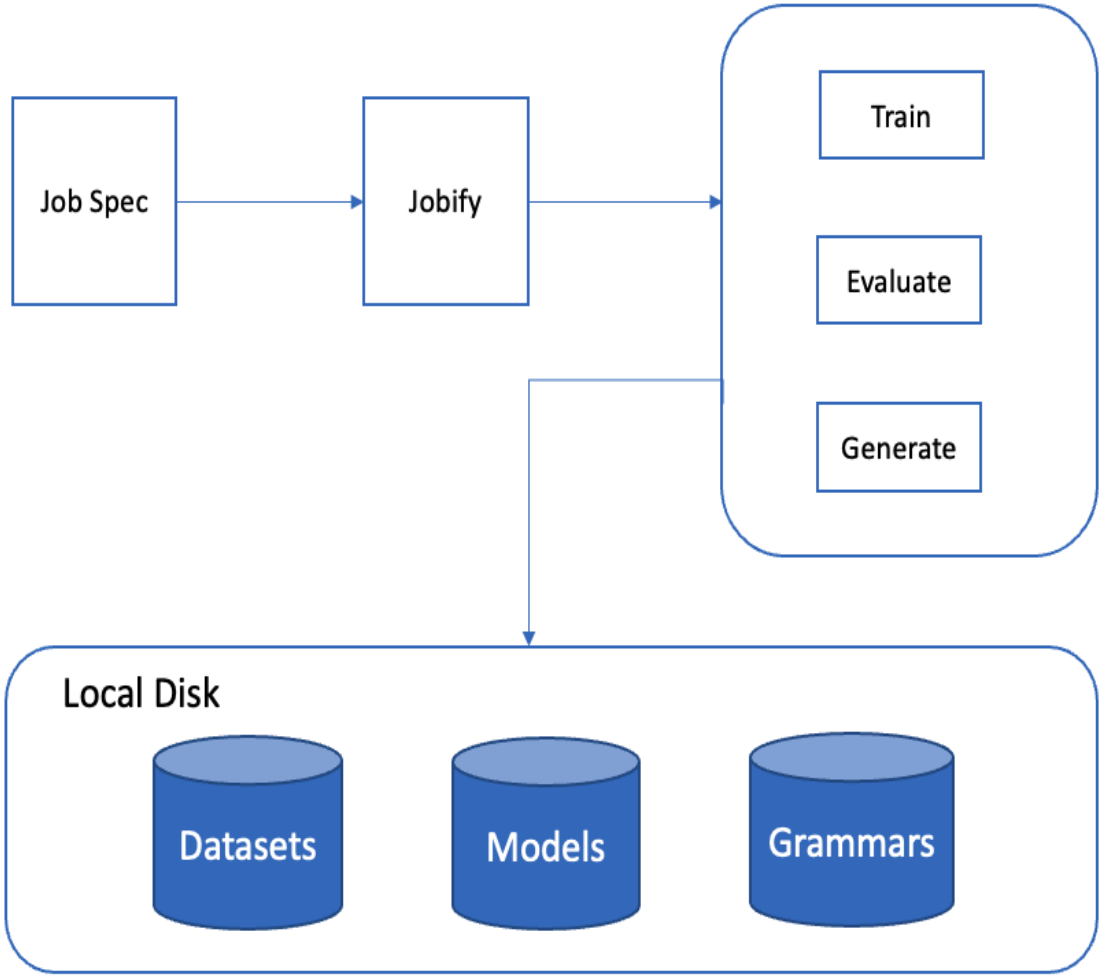

Figure 1-7: DEG2.0 Local

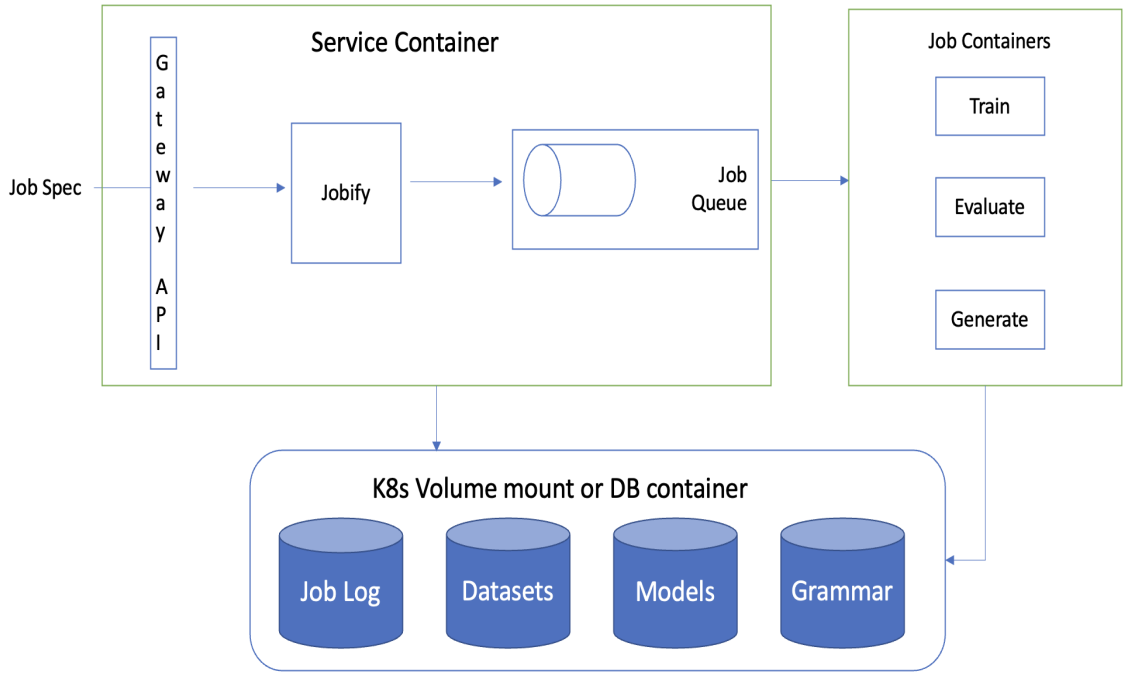

Figure 1-8: DEG2.0 Remote

1.6.3 Jobs

As we saw in section 1.4, DEG has three main responsibilities: the learning of grammars, the evaluation of grammars, and the generation of molecules from learned grammars. In DEG2.0, we restructured the codebase such that these three responsibilities could be accomplished with individual functions, `train()`, `evaluate()`, and `generate()` (note `train()` is the equivalent to DEG's `learn()`). In principle, each function should only worry about completing its specific task, so the logic of how to run its task should be determined elsewhere [18].

For example, if a user wants to generate molecules using a grammar, they should be able to call `generate(grammar, num_samples)` where `grammar` is some learned grammar object and `num_samples` is the number of molecules to generate. This seems simple enough, but there are a number of issues with this approach. Making this the primary way of interacting with the library means the user would have to write actual Python code, initializing a grammar object, passing it to the function, and deciding what to do with the outputs. We could solve one of these problems by loading the grammar within the function, having the user specify a file name

which would only require typing a string into a function. This would be one level of abstraction higher when compared to initializing the object, so we would improve on that front. However, this approach means the `grammar` input would have to change to a file name and `generate()` would have the additional responsibility of reading from a file which is not a natural requirement for generating molecules. Also, part of the algorithm for learning grammars involves generating and evaluating molecules. It would be inefficient if the learning algorithm saved the learned grammar into a file to then load it within the generate function. The functions `generate()` and `evaluate()` should be kept modular enough to function within the learning algorithm and also for handling client requests without needing to duplicate any code. To do this, we made sure the functions only took in what they truly needed as parameters and only outputted the natural result of their calculations.

The examples above illustrate the trade-offs between a few competing design principles. We wish to abstract as much of the business logic as we can from the client, maximize the flexibility in interacting with the library, and keep the business logic as modular as possible. To deal with these trade-offs, we chose to introduce the concept of "jobs." In addition to the `train()`, `evaluate()`, and `generate()` functions, we created train, evaluate, and generate jobs.

Referring back to the DEG2.0 package diagram, we see the code for different jobs lies in the `jobs` package within `deg`. For each job, we created an individual script `jobs/train.py`, `jobs/evaluate.py`, and `jobs/generate.py`. In each of these scripts we have a class called `$JOBTYPESpec` and a function called `run_$JOBTYPE_job()` where `$JOBTYPE` is either "train", "evaluate", or "generate", depending on the script it is defined in. `$JOBTYPESpec` extends a base class called `JobSpec`. This is an abstraction that promotes scalability when it comes to spec checking, which we talk about in the next section. `run_$JOBTYPE_job()` gathers the necessary data, converts it to the right format, and runs the individual `train()`, `evaluate()`, or `generate()` function. Each `run_$JOBTYPE_job()` function is registered to a `JOB_REGISTRY` which is implemented in the `jobify` package. Registries (which we talk about in detail in 1.6.5) also promote scalability. Train, evaluate, and generate jobs are all jobs, so they

all have a function that runs them. As mentioned in 1.6.2, a spec gets turned to a job and the job is run. With a registry, the specific `run_$JOBTYPE_job()` function to run can be found from the job type given in the spec. If new jobs need to be created in the future, developers can simply focus on the code for the individual job without needing to touch the code that runs the job from the spec. In other words, they can focus on 'what' to run rather than 'how' to run. Also, the code that runs the jobs does not need any conditioning. As long as the jobs are registered, the code dynamically gets the correct `run_$JOBTYPE_job()` function from the job type in the spec and runs it. This can be one line of code instead of many `if/else` statements conditioned on the job type to decide which `run_$JOBTYPE_job()` to run.

Within each job script, there are three responsibilities: defining the spec checker, defining `run_$JOBTYPE_job()`, and running one of the main functions. The following code snippet shows an outline for the code in each of the job scripts:

```
@jobify.register_job_spec($JOBTYPE)
class $JOBTYPESpec(JobSpec):
    # define the spec outline and spec validation functions


@jobify.register_job($JOBTYPE)
def run_$JOBTYPE_job(spec: $JOPTYPESpec, io_manager: IOManager) ->
    $JOBTYPEResult:
    # initialize/prepare necessary parameters
    results = $JOBTYPE(parameters)
    # log results if needed
```

We talk about spec checking in 1.6.4, so below we only mention the details in each `run_$JOBTYPE_job()` function.

Train Job

The train job is responsible for running `train()`. This is what the declaration of `train()` looks like:

```
def train(
    agent: Agent,
    feature_extractor: GNNFeatureExtractor,
    metric_specs: List[TrainMetricSpec],
    training_data: Path,
    hparams: HyperParameters,
    save_history: bool = False,
    save_artifacts: bool = True,
    artifact_callbacks: Optional[List[Callable]] = None,
    metrics_callbacks: Optional[List[Callable]] = None,
    **kwargs,
) -> TrainResult:
```

Referring back to the API, we know that all the user provides is a spec. `run_train_job()` is responsible for filling in the parameters needed by `train()` from the spec. It does this by initializing the `agent` and the `feature_extractor` from the details in the spec and defining the callback functions. It also has a check for whether this train job should resume from a checkpoint. If so, it initializes the `agent` weights to the weights from that checkpoint. Finally, the train job saves the results from `train()` if prompted to do so in the spec.

Evaluate Job

The evaluate job is responsible for running `evaluate()`. The `evaluate()` declaration looks like this:

```
def evaluate(metrics: List[Metric], data: Dict[str, Any]) ->
    List[MetricResult]:
```

`evaluate()` needs a list of metrics and a data dictionary. The evaluate job takes the metrics given in the specs and initializes them, putting them into a list. The data dictionary is contains different information depending on what metrics the user wants to run. Metrics may need molecule samples, grammar objects, or neither. Having

a data dictionary allows developers to define metrics assuming all the information needed to run the metrics will be gathered by `run_evaluate_job()` and passed to `evaluate()` within `data`. Same as above, the evaluate job saves the results from `evaluate()` if told to do so.

Generate Job

The generate job is responsible for running `generate()`. The `generate()` declaration looks like this:

```
def generate(
    grammar: ProductionRuleCorpus,
    num_samples: int,
) -> List[str]:
```

`generate()` needs a grammar object and a number of samples. The number of samples can come straight from the spec but the grammar object needs to be initialized within `run_generate_job()` before passing it to `generate()`. The generate job can also save the results from `generate()`.

1.6.4 Specs

We have mentioned the spec many times already without fully explaining what it is. In this section, we define what specs are, motivate them, and provide examples.

Formart

A spec is a JSON or YAML file containing all the information necessary for running a job. DEG2.0 supports both JSON and YAML, but to avoid repeating ourselves, we focus the conversation on JSON. The reason we use JSON is because JSON objects are serializable, meaning they can be converted to a series of bytes which can be transmitted across a network. This makes JSON the standard request-response data type for RESTful APIs [5]. JSON objects get serialized into bytes when sent as requests and deserialized from bytes when received as responses. This is easy to do in

Python with the built-in package `json`. JSON is also human-readable which makes defining and interpreting specs easy for human users. Specs can be defined as Python dictionaries too, as long as the dictionaries can be converted to JSON. When defining specs as dictionaries, the specs can only contain the following basic Python objects to ensure they can be converted: `dict`, `list`, `tuple`, `str`, `int`, `long`, `float`, `True`, `False`, `None`. Custom objects such as a `Grammar` object cannot be converted to JSON, so if jobs need custom objects to run, they must build them in the back-end based on the information given in the spec, as was described in the previous sections.

JSON is not the only way to serialize data. Custom Python objects can be sent over a network by using the third party `pickle` package [10]. This package serializes and deserializes objects by "pickling" and "unpickling" them. There are a few reason why we chose JSON over Pickle. The first reason is that Pickle only makes sense if the user wants to send and receive custom objects. We did not design for this possibility for two reasons: it assumes the user knows how to program in Python and it requires having additional files in the client to initialize the objects. The former is not an assumption we should make when minimizing the difficulty in using a library. The latter is less obvious, but one can imagine the user wanting to send the `agent` neural network rather than the instructions for making it. In this case, the PyTorch code that defines the neural network would have to live in the client. This is undesirable because clients should be lightweight and the code needed to run jobs should be protected from user modifications. We can make the same argument for not initializing other objects such as grammars in the client. The second reason for not using Pickle is that it is not secure. One can construct malicious pickle data which will execute arbitrary code during unpicking. We do not want users to have the power of running arbitrary code in the back-end since this could break the system.

I want to quickly note that the arguments above are not very strong in the context of the local version. In the local version, all of DEG2.0 is in the same machine as the client, so the comments on security do not apply here. The user could change anything in the code if they wanted. However, it is still valuable to design the local version with these things in mind because they matter for the remote version. If

you have many users using the same server and one of them breaks the server, then the application stops working for all the users. Also, these patterns lower the client's complexity and establish a workflow that does not require dealing with business logic, which is desirable based on the customer's requirements.

Spec Types

Same as jobs, there are three types of specs: train, evaluate, and generate. When the user defines a spec, they are defining what type of job they want to run and what parameters they want the job to run with. The subsections for 1.6.3 showed the parameters each of the primary functions take. In the specs, users can define what they want these parameters to be. This means users can run the same job in various ways by simply changing the spec definition. And since JSON is readable and can be made from normal text, users have the option of not touching any Python code to run jobs. They could simply write a JSON file and use the command line:

```
degrammar train --config configs/train_spec.json
```

In this example, the user would be running a train job using the spec defined in `train_spec.json`. Another way to define specs is to make Python dictionaries. If the user opts to do it this way, they can define the spec in a Jupyter notebook, turn it to a JSON string, and send it to the back-end.

```
import json
spec = dict{"my_custom_job_spec": data} # not a real spec
json_string = json.dumps(spec)
# send to back-end
```

Spec Examples

We provide default specs that the user can use for running canonical jobs. Below we see examples for the three different specs:

```yaml
job_type: train
job_name: my_custom_train_job
training_data: path/to/training_data.txt
metric_specs:
    - name: diversity
    - name: retro_star
hparams:
    epochs: 1
    learning_rate: 0.001
    optimizer: Adam
    num_generated_samples: 2
    num_mcmc_samples: 1
    gamma: 0.99
    motif: false
agent:
    - type: Linear
    kwargs:
        in_features: 302
        out_features: 128
    - type: Dropout
    kwargs:
        p: 0.5
    - type: ReLU
    kwargs: {}
    - type: Linear
    kwargs:
        in_features: 128
        out_features: 2
    - type: Softmax
    kwargs:
        dim: 1
gnn_model_path: /path/to/model.pth
save_artifacts: true
save_history: false
```

Figure 1-9: Train Spec

```yaml
job_type: evaluate
job_name: my_custom_evaluate_job
grammar_path: path/to/grammar.json
samples: path/to/samples.txt
metric_specs:
    - name: diversity
    - name: retro_star
save_artifacts: true
```

Figure 1-10: Evaluate Spec

```yaml
job_type: generate
job_name: my_custom_generate_job
num_samples: 10
grammar_path: path/to/grammar.json
save_artifacts: true
```

Figure 1-11: Generate Spec

There are many parameters in each spec that the user can define to create custom jobs. Our customer wanted more flexibility in their experiment workflow. With specs, changing the evaluation metrics in the training algorithm is as simple as adding or deleting lines of text under the `metric_specs` field. The `agent` neural network can be fully determined from the spec. If users want to test performance across different networks, they can just create a few different specs with different parameters and run those jobs. Before, they would have needed to edit metrics and network parameters within the code.

To further simplify things, the user does not need to provide all of the values seen above when defining specs from scratch. For example, if job names are not provided, DEG2.0 creates a unique job name and returns it in the response. If the hyperparameters for `agent` are not provided in the train spec, the train job initializes `agent` with default parameters. Users should only need to specify configurations that differ from the defaults to avoid unnecessary typing. Here are examples of the simplest specs the user could provide:

```
job_type: train
training_data:
    path/to/training_data.txt
```

```
job_type: evaluate
```

```
job_type: generate
num_samples: 10
grammar_path:
    path/to/grammar.json
```

Every value not directly specified in these short specs has a default value written in the spec object definitions (details below).

Spec Checking

So far we have been talking about specs as JSON files or objects. In the client, specs are no more than that. However, once specs are received in the back-end, they are turned into spec objects. We have three main spec objects: `TrainSpec`, `EvaluateSpec`, and `GenerateSpec`. These are all defined within the `jobs` package in the files `train.py`, `evaluate.py`, and `generate.py`, as we saw in the 1.6.3 section. These three classes are abstracted by a parent class called `JobSpec` which extends a class called `BaseModel`. This means that the three spec classes are all instances of `BaseModel`. `BaseModel` is a class from a third-party Python packaged called `pydantic` which we use for spec checking. Pydantic is widely used in industry for data validation and settings management using Python type annotations [7]. It enforces type hints at runtime and provides user friendly errors when data is invalid. By inheriting from `BaseModel`, we can use Pydantic types and validators for ensuring specs are input correctly by the user.

Type Checking At runtime, spec objects are built from the inputted specs. The spec objects have attributes corresponding to the different elements in the spec. These attributes are tied to Pydantic types. For example, the `job_name` of each job has to be a `str`, so it is given the Pydantic type `StrictStr`. If the user were to send a spec that has an `int` for `job_name`, Pydantic would return an error telling the user that `job_name` has to be a `str`. Pydantic was designed to be intuitive, so seeing the actual code for how we define the main spec objects may be helpful for understanding how

the checking works:

Job Spec :

```
class JobSpec(BaseModel):
    job_name: StrictStr = Field(default_factory=lambda: generate_slug(2))
    job_type: Optional[StrictStr]
    job_id: StrictStr = ""
    ts: Optional[datetime] = Field(default_factory=datetime.now)
    save_artifacts: StrictBool = True
```

Train Spec Object :

```
class TrainSpec(JobSpec):
    job_type: Literal["train"] = "train"
    training_data: Path
    hparams: HyperParameters = Field(default_factory=HyperParameters)
    agent_spec: List[Dict] = Field(
        default_factory=lambda: [
            {
                "type": "Linear",
                "kwargs": {
                    "in_features": 302,
                    "out_features": 128,
                },
            },
            {"type": "Dropout", "kwargs": {"p": 0.5}},
            {"type": "ReLU", "kwargs": {}},
            {"type": "Linear", "kwargs": {"in_features": 128,
                "out_features": 2}},
            {"type": "Softmax", "kwargs": {"dim": 1}},
        ]
    )
```

```
    gnn_model_path: Path = "./models/GCN/supervised_contextpred.pth"
    resume_path: Optional[Path] = None
    metric_specs: List[TrainMetricSpec] = Field(
        default_factory=lambda: [
            TrainMetricSpec(name="diversity", weight=1),
        ]
    )
    save_history: StrictBool = False
```

Evaluate Spec Object :

```
class EvaluateSpec(JobSpec):
    job_type: Literal["evaluate"] = "evaluate"
    job_name: StrictStr = Field(default_factory=lambda: generate_slug(2))
    grammar_path: Optional[
        Path
    ]
    samples: Optional[Union[Path, List]]
    metric_specs: List[EvaluateMetricSpec] = Field(
        default_factory=lambda: [
            EvaluateMetricSpec(name="diversity"),
        ]
    )
```

Generate Spec Object :

```
class GenerateSpec(JobSpec):
    job_type: Literal["generate"] = "generate"
    num_samples: PositiveInt
    grammar_path: Path
```

Looking at these spec object definitions, one can quickly see the overall pattern:

```
attribute : PydanticType = default_value
```

This pattern is the same for all of the type checking. We also take advantage of Pydantic's functionality by defining custom types. The custom types are simply new classes that extend `BaseModel`, same as the main spec objects. The reason we say *main* spec objects is because these custom types are spec objects too, but they are only used for type checking within the main spec objects. The custom types let us make more complex checks than the `StrictStr` example we mentioned for `job_name`. For example, in `TrainSpec` the `metric_specs` attribute has to be a list of `TrainMetricSpec`, a custom Pydantic type that does the spec checking for metrics. The `hparams` attribute is another example of this, mapping to the `HyperParameters` type. Defining custom types lets us implement multiple layers of spec checking while keeping the code readable and modular.

Validators For more sophisticated spec checking, Pydantic offers the ability of defining validation functions. Since we only have a few of these, we can mention all of them. Within `TrainSpec` we have `validate_metric_specs()`, `validate_hparams()`, and `validate_agent()`. `validate_metric_specs()` makes sure that train jobs that want to use the Diversity metric have a value greater than 1 for `hparams.num_generate_samples`. If the user tries to run a train job using the Diversity metric but also sets `hparams.num_generated_samples` < 2, then the back-end will send a `MetricSpecError` to the client saying "there must be more than 1 sample provided for 'diversity' metric to work". `validate_hparams()` ensures the specified optimizer to use for `agent` is valid, otherwise an `OptimizerSpecError` is returned. `validate_agent()` ensures the given neural network parameters can yield a valid PyTorch module. This function does a lot of checks so we will not mention all of them. Some of the checks make sure the user inputted valid names for the PyTorch modules and module layers, for example. Any errors are returned as `AgentSpecErrors` with corresponding messages. Within `EvaluateSpec` we have another version of `validate_metric_specs()`. It makes sure that if the evaluate job wants to run a Diversity, Num Samples, or Retro Star metric, a list of samples must be provided. Same as in the train spec, if running

the Diversity metric, the sample list must have more than one sample. And finally, if running the Num Rules metric, a grammar must be specified. The reason we have separate metric validators for the evaluate spec and the train spec is because there are some guarantees about the data available at runtime when running a train job that we do not have for evaluate jobs. For example, `train()` will always make a grammar, so the grammar does not need to be specified by the user in the train spec for the Num Rules metric to run. Lastly, `GenerateSpec` does not have any validators.

Remarks Using Pydantic types and validator functions for spec checking is another example of a decision made to facilitate the user workflow and improve the maintainability of the library. Spec checking provides a clear contract for client-server communication. Specs have to be inputted following the rules given in the spec object definitions. If not, concise errors are returned letting the user know what to change. If more validator checks need to be added in the future, we have an array of custom error types as shown above which can be reused. Again, this makes the code more readable and scalable.

1.6.5 Metrics

In this section, we talk about another important piece of refactoring done in DEG2.0. We have alluded to metrics many times already. The role of metrics in DEG is to evaluate grammars while they are learned. They make up the objective function of the learning algorithm. It is worthwhile to see how metrics were implemented in DEG to get a better understanding for why we refactored them. Here is the block of code where metrics live in the original codebase:

```
for _metric in metrics:
  assert _metric in ['diversity', 'num_rules', 'num_samples', 'syn']
  if _metric == 'diversity':
      diversity = div.get_diversity(generated_samples)
      eval_metrics[_metric] = diversity
  elif _metric == 'num_rules':
```

```
        eval_metrics[_metric] = grammar.num_prod_rule
    elif _metric == 'num_samples':
        eval_metrics[_metric] = idx
    elif _metric == 'syn':
        eval_metrics[_metric] = retro_sender(generated_samples, args)
    else:
        raise NotImplementedError
```

Concerns This code is found within DEG's `evaluate()`. There are a number of things to note here. First, the logic involves `if/else`. This is undesirable because it makes the code less maintainable. Each time a new metric is created, a new condition must be added. Also, using `if` and `elif` means that only one metric can run at a time which could be a limitation. Second, there is no shared structure between metrics. Even though Diversity, Num Rules, Num Samples, and Syn are all "metrics," each one of them runs differently. Diversity uses a function from an object called `div`, Num Rules is computed in-line using a grammar, Num Samples (not obvious from just this snippet of code) uses a variable that is summed while molecules are generated within `evaluate()`, and Syn calls a function that writes to the sender file we described in section 1.4 and reads from the receiver file. Third, there is no template to follow to create new metrics. One could basically put anything into the `eval_metrics` list without a warning in the IDE. The code would have to run before realizing something was coded incorrectly. Lastly, there is no way for a user to run metrics other than creating the `metrics` list in the code.

Solutions We address all of these points with the `metrics` package of DEG2.0. Taking inspiration from the Command Design Pattern [34], we define all the metrics in `metrics.py`. The Command Design Pattern is a behavioral design pattern in which an object is used to encapsulate all the information needed to perform an action or trigger an event at a later time. We define an abstract class called `Metric` which takes an unspecified number of keyword arguments in the constructor and has one function called `compute()`. `compute()` takes in `data` and returns a `MetricResult`.

```
class Metric(ABC):
    def __init__(self, **kwargs):
        pass

    @abstractmethod
    def compute(self, data) -> MetricResult:
        pass
```

This class is the equivalent of the "Command Interface" in the Command Design Pattern. We extend this class with individual classes for all of the metrics mentioned above, naming them `DiversityMetric`, `NumRulesMetric`, `NumSamplesMetric`, and `RetroStarMetric` (instead of Syn). Placing the `@abstractmethod` decorator above `compute()` means that all classes that extend `Metric` must implement `compute()` with the same parameters and return type. Here is an example of the code for a simple metric, `NumSamplesMetric`:

```
@register_metric("num_samples")
class NumSamplesMetric(Metric):
    def compute(self, data):
        return MetricResult(name="num_samples",
            value=len(data["smiles_samples"]))
```

This metric does not need any keyword arguments to set attributes, so we can use the parent constructor. With this design, we have created a template and a shared structure among metrics, solving the second and third concerns listed above. All new metrics must extend the parent `Metric` class and implement the `compute()` function. When evaluating grammars, we should be able to just run `compute()` for each metric instead of doing different things for each. To see how this design solves the first and last concerns, we need to talk about the decorator `@register_metric`.

Registries We briefly mentioned registries in section 1.6.3. Registries allow us to use the Python Registry Pattern [31]. This pattern is useful in several situations. An example situation is when you want to streamline a factory class (see Factory

Design Pattern [33]), which takes an input label, like the name of a class, and creates an object of that type. This is exactly the situation we have. Ideally, we could use the metric specs given in the job specs to create metric objects when needed. Metric specs contain the names of the metrics the user wants to use in a train or evaluate job. Adding the decorator `@register_metric(metric_name)` to a metric class puts that class into the `METRIC_REGISTRY`. We can use this object to map `metric_name` to metric class. We do the fetching of metric classes from the registry with a function called `get_metric()`:

```
def get_metric(metric_name: str) -> Type["Metric"]:
    if metric_name not in METRIC_REGISTRY:
        raise ValueError(f"Metric {metric_name} not found in registry.")
    return METRIC_REGISTRY[metric_name]
```

And the creating of metrics in a factory function called `create_metric()`:

```
def create_metric(metric_spec: MetricSpec) -> Metric:
    metric_cls = get_metric(metric_spec.name)
    return metric_cls(**metric_spec.dict())
```

With `get_metric()`, `create_metric()`, and the `METRIC_REGISTRY`, we resolve the first and last concerns. Now, whenever we run `evaluate()`, we pass it a list of metrics created by running:

```
metrics = [create_metric(m) for m in spec.metric_specs]
```

Then, inside `evaluate()`, we run `compute()` for each of these metrics. This design simplifies not only the block of code used for DEG's metric implementation, but the entirety of the old `evaluate()` function into just this one line:

```
return [metric.compute(data) for metric in metrics]
```

There is no longer a need for `if/else` nor for editing the `evaluate()` function. This makes the code more maintainable for developers since they can focus on adding new metric classes rather than editing business logic and easier to use for users since they

can just choose what metrics to run within the spec rather than within the code.

1.6.6 Data Logging

All the refactoring done for DEG2.0 would be useless without an easy way to see job results. In DEG, the Python `logging` package [11] was used to output training results to the console. Though this is simple, it may not encapsulate everything a user might want to see from an experiment. It is also not persistent, so if users wanted to check on an experiment they ran in the past, they would not be able to. It would be up to them to save results into files or databases and to remember the configurations they used for different experiments.

IOManager

We implement the `io_manager` package to deal with this. This package has a file called `io_manager.py` which defines an abstract class named `IOManager` with a number of abstract methods. Here is a UML representation for what the class looks like:

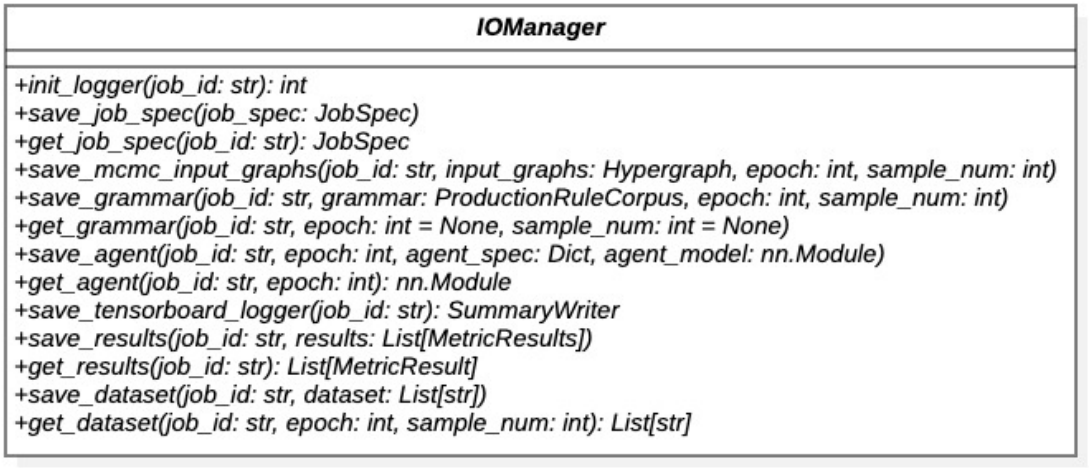

Figure 1-12: IOManager Class

As seen in section 1.6.5, defining an abstract class as a template for other classes makes the code readable, maintainable, and simple. This class is extended by `FileIO-`

`Manager`, defined within `file.py` also in the `io_manager` package. `FileIOManager` is an `IOManager` which implements all of the abstract methods such that logging happens on the local disk. Following this pattern, developers could define a `DatabaseIOManager` in a file called `database.py` to implement the abstract methods for logging to databases. Since `FileIOManager` and `DatabaseIOManager` are both `IOManagers` implementing the same methods, they can be used interchangeably in the rest of the code. Though we only implemented `FileIOManager`, we can see how the structure of this package is designed to quickly transition into a remote version using databases.

Artifacts

As shown in the code snippet in section 1.6.3, each job's `run_$JOBTYPE_job()` function takes in an `io_manager` which is an `IOManager` object. Depending on the type of job, a number of artifacts may be generated and the user has the option whether to store those artifacts or not. Artifacts are just the outputs of a job. These options are provided in the job spec, so each job can determine what calls to make with the `io_manager` by parsing the spec. Here we list the artifacts each job could log:
Train Artifacts: job spec, logs, agent network weights, grammar checkpoints, checkpoints for MCMC sampler input graph, Tensorboard logs for policy loss and grammar evaluation per epoch [1]
Evaluate Artifacts: job spec, logs, evaluation results
Generate Artifacts: job spec, logs, generated molecules

Excluding the job spec and the run log, the user has the option to turn off artifact logging for each job. For an evaluate or generate job, the user may specify a boolean parameter called `save_artifacts` in the spec to turn the artifact logging on or off. For a train job, the user has the option to either store all of the artifacts generated in `train()` or just opt for the best ones (best agent weights, best grammar). We do this to maximize flexibility. If the user is running quick experiments, they may not want to waste disk space saving all of the artifacts of a train job. The relevant configurations are shared below in figures 1-13 through 1-15.

```
train_spec = {
    ....
    "save_artifacts": True
    "save_history": True
}
```

Figure 1-13: All Artifacts Across Training

```
train_spec = {
    ....
    "save_artifacts": True
    "save_history": False
}
```

Figure 1-14: Best Artifacts Across Training

```
train_spec = {
    ....
    "save_artifacts": False
    "save_history": False
}
```

Figure 1-15: No Artifacts

File Structure

For this release, we did not implement databases for the remote version, so all the logging happens on the local disk. The artifacts are stored inside a folder named `runs` by default. The file structure in this folder is the following:

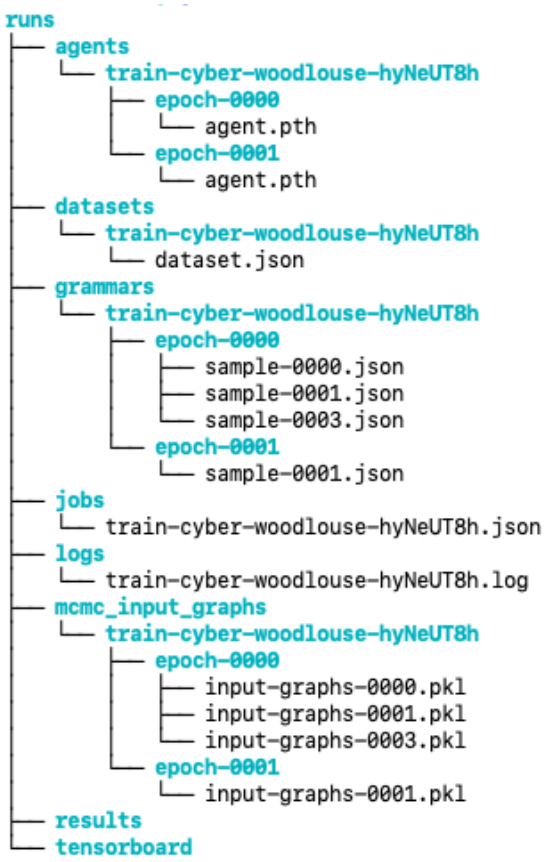

Figure 1-16: File Structure

`train-cyber-woodlouse-hyNeUT8h` is the train job that was run. We can see it was run for two epochs. Each folder within `runs/` has sub-folders for each job that

created and saved that specific artifact. This way, users can search for the artifacts they need and filter by job. Also, if they need to reference the spec used to run that specific job or the logs created at runtime, they can find them in `runs/jobs/` and `runs/logs/`.

1.6.7 Jobify

We have now talked about the main components in the design of DEG2.0, so we can finally take a step back and talk about the details of the package that brings them all together, `jobify`. Jobify is the package that actually creates and runs jobs. The workflow, as mentioned in section 1.6.2, starts from a spec which `jobify` receives, converts to a `JobSpec` object, and runs the job. Here, we share the details behind how this works.

Recall the two ways of running jobs in DEG2.0 are through the command line or through a Jupyter notebook. The command line call makes a call to a function within `jobify/jobs.py` named `run_job()`. This function takes in a `spec` and a `run_dir`. The `spec` is either a dictionary of a file path and `run_dir` is a string specifying the root folder for logging (default values is "runs/"). Running through Jupyter works similarly, one would just need load the spec from a JSON or YAML file or from a dictionary defined within the notebook and run the `run_job()` function.

We have been making this distinction between the front-end and the back-end throughout much of the thesis without being specific about where the separation lies within the code. Although there is technically no real separation in the local version given all the code lives in the same machine, the library is still designed for separation. And we can finally point at it. In the remote version, we can place all of the code we have described so far into a server and give the client a single responsibility, making specs. These specs would get sent as HTTP requests to the server, where they are deserialized into Python dictionaries that get passed to `run_job()`. This makes `jobify` the entry point for the server side. All the client side would need is the ability to make JSON objects and send HTTP requests. Though we would have to make a few other changes, such as extending `IOManager` with a remote logging class,

the general workflow between the finished local version and the projected remote version is virtually identical, meaning our design works.

Back to Jobify. `run_job()` creates the job spec object from the spec and makes a `JobRunner` object called `job_runner`. It uses the job type given in the spec to index into the `JOB_SPEC_REGISTRY` and get the corresponding job spec class. Then it uses the class to make either a train, evaluate, or generate job spec object. During the initialization of the spec object, Pydantic runs all of the spec checking and validation and returns errors if the spec was specified incorrectly. Then, `run_job()` runs the line `job_runner.run(job_spec)`. This function uses the job type to get the corresponding `run_$JOBTYPE_job()` function with the following line `job = get_job(job_spec.job_type)`. Then it does a few things to initialize the logging and concludes with the simple line `return job(job_spec, self.io_manager)` which actually runs the job and returns its results.

Notice how the design of DEG2.0 enabled making a `jobify` package that creates and runs jobs without a single job type check. Because of the things we have talked about, job design, spec design, spec checking, metrics, registries, loggers, Jobify will work for any job you give it, even new ones that could be created in the future, without needing to change any code. This adheres to several software engineering principles we have been alluding to such as the open-closed, single-responsibility, and separation of concerns principles [13].

1.6.8 Engineering Choices

Multi Processing

The Retro Star metric is the heaviest metric in DEG2.0 so far. As mentioned in section 1.4, it was implemented using file sharing and separate shell processes. This is not ideal for a number of reasons. Running Retro Star in separate shell processes means that each process needs a copy of the model which leads to each one taking around 5Gb of memory. This creates a bottleneck when running jobs using the Retro Star metric locally, since local computers have relatively small memory limits. Also,

file operations (lock, unlock, read, and write) are slow compared to operations done in memory. Lastly, all of these individual processes had to run in different terminal windows and had to be cleaned up manually after execution, which is cumbersome for users to do.

To deal with these issues, we explored two solutions, `Celery` [35] and `multiprocessing` [12]. Celery is an asynchronous task queue or job queue based on distributed message passing. Task queues are used as a mechanism to distribute work across threads or machines. Their input is a unit of work called a task. Dedicated worker processes constantly monitor task queues for new work to perform. Celery communicates via messages, usually using a broker to mediate between clients and workers. To initiate a task, the client adds a message to the queue and the broker delivers that message to a worker. A Celery system can consist of multiple workers and brokers, giving way to high availability and horizontal scaling. This tool sounds quite similar to the way our API is set up, with a client sending messages (job specs), a broker dealing with them (Jobify), and workers running them (`train()`, `evaluate()`, and `generate()`). This made Celery an attractive option to run jobs, specially for jobs including the Retro Star metric given Celery's ability to distribute tasks across multiple workers.

Multiprocessing is a package that supports spawning multiple Python processes. It offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock [30] by using subprocesses instead of threads. It has a `Pool` object which offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism). Using the `Pool` object, we can solve the problem of needing to copy the Retro Star model across processes, since they can share the model. We tried both approaches and compared their performance, shown in table 1-1. We can see that the best performing option is multiprocessing. The issue with Celery is that it is designed for distributed workflows. This means it would be a great option for the remote application but proves to be too heavy for the local version. Running Celery on the local version requires creating a Redis server to communicate with the Celery client. This commu-

Samples	Multiprocessing (sec)	File Handling (sec)	Celery (sec)
5	49	97	123
10	73	107	156
20	168	253	295
40	350	392	582
80	525	559	795
160	1381	1366	1681
320	2678	2829	2988

Table 1.1: Time to run Retro Star across different sample amounts using 5 processes

nication is slow and not necessary if the data is all on the same machine. Showing that Celery can be used and that it fits well with our API was a good proof-of-concept for the remote version, but for the local version, we chose multiprocessing.

Poetry

In section 1.5.3 we mentioned the need for two separate Conda environments for installation of DEG and a few installations from source. This is undesirable as it slows down the setup process and introduces additional complexity for the user. We leverage Poetry, a tool for dependency management and packaging in Python, to solve these issues [28]. Using Poetry we can declare the libraries our project depends on and Poetry will take care of managing them (installing and updating the correct versions) and configuring the environment. The list of dependencies can be found within `pyproject.toml`. After DEG2.0 is cloned into one's computer and Poetry is installed, setting up the environment is as simple as running these two commands in the root directory of the project:

```
poetry install
poetry shell
```

1.6.9 Remote Version

We have mentioned the remote version many times in the prior sections, motivating design and implementation choices with the idea that code should be reused between

the local and remote versions without large architecture changes. Although the remote version was not fully completed within my time at the lab, we were able to deliver a release that could run the Retro Star metric remotely in a Docker container [9]. The only additional work this required was setting up a Docker container and connecting to it, validating our design. We only focused on containerizing the Retro Star metric since this is the bottleneck of the training process in DEG2.0. Jobs that do not run Retro Star are relatively light-weight, making it less valuable to containerize them for the first release of the library.

1.7 Future Work

There is plenty of work that could be done in the future to improve DEG2.0. An immediate next step would be to complete the remote version, containerizing all jobs and polishing the workflow. Other next steps would be informed by the feedback from the customer company. There is a chance they love what we have done and would like to keep things as they are, but there is also a chance they end up needing significant changes. In any case, as they scale their work, they will likely want to see support for more metrics, more datasets, different types of logging, other types of generative models, tools to automate comparison and analyses of different models, a web application for the client, capacity to distribute jobs among many servers, and a variety of additional features that could make DEG2.0 more capable. Unfortunately, my time at the lab was short-lived, so I could only form part of the first iteration of DEG2.0. It will be nice to see what developments happen in the future.

1.8 Generalizing to Other Domains

We allude to the secondary goal of generalizing to other domains many times throughout the thesis. Due to time restrictions, we chose to focus on only one domain, generating molecules, to ensure we could deliver a product to the customer before my internship was over. However, as mentioned throughout our work, this secondary

goal influenced many of the decisions we made. Much of the work behind creating abstractions and maintainable code went hand-in-hand with the secondary goal. This effort proved to be valuable in the months following my internship. My former teammates Aditya and John, along with the researchers behind DEG, began a new project focused on using circuit data to train a grammar to generate circuits. The thought process, abstractions, and code developed during the DEG2.0 project have been used to facilitate the transition to this new domain. This project is currently being worked on in the lab and has shown promising results.

1.9 Conclusion

Our first goal for this project was to make an easy-to-use, capable library with a standard API that could work locally and remotely. Our second goal was to generalize the library as much as possible. Throughout the thesis, we have given many examples of choices made to make the library easier to use and understand, capable of running diverse experiments, and transferable between local and remote versions. Each of these choices has considered the second goal, which is shown by the abstractions and modularity created in both the architecture and the implementation of DEG2.0.

Though it is difficult to quantity objectives such as ease-of-use and generalization, we provide proxies for these objectives by alluding to customer requirements, software engineering principles, and common AI workflows. Rather than providing hard numbers for our improvements, we support them through examples such as plug-and-play metrics, and through comparisons to DEG such as the ability for persistent logging which was previously nonexistent.

Overall, this has been an amazing project to work on. I have learned a great amount about software engineering for AI applications and am glad I was able to do it for a real-world use case. Once again, thank you to MIT, IBM, and to everyone who has supported me during these unforgettable 6 years as an MIT student.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Anaconda.

[3] Evonnbsp; Dong Bing. Key ideas about separation of concerns, Sep 2020.

[4] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale gan training for high fidelity natural image synthesis, 2018.

[5] Kolade Chris. Rest api best practices – rest endpoint design examples, Sep 2021.

[6] Kolade Chris. Open-closed principle – solid architecture concept explained, Feb 2023.

[7] Samuel Colvin, Jun 2017.

[8] Hanjun Dai, Yingtao Tian, Bo Dai, Steven Skiena, and Le Song. Syntax-directed variational autoencoder for structured data, 2018.

[9] Docker. Accelerated, containerized application development, Apr 2023.

[10] Python Software Foundation. Pickle - python object serialization.

[11] The Python Software Foundation. Logging - logging facility for python.

[12] The Python Software Foundation. Multiprocessing - process-based parallelism.

[13] Erich Gamma. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 2016.

[14] Dedre Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7(2):155–170, 1983.

[15] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[16] Minghao Guo, Veronika Thost, Beichen Li, Payel Das, Jie Chen, and Wojciech Matusik. Data-efficient graph grammar learning for molecular generation, 2022.

[17] Red Hat. What is an api?, Jun 2022.

[18] Thorben Janssen. Solid design principles: The single responsibility explained, May 2023.

[19] Jamie Juviler. Rest apis: How they work and what you need to know, Aug 2022.

[20] Hiroshi Kajino. Molecular hypergraph grammar with its application to molecular optimization, 2018.

[21] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[22] Andrew Koenig-Bautista. What is inheritance? a simple oop explanation, Dec 2019.

[23] Mario Krenn, Florian Häse, AkshatKumar Nigam, Pascal Friederich, and Alan Aspuru-Guzik. Self-referencing embedded strings (SELFIES): A 100% robust molecular string representation. *Machine Learning: Science and Technology*, 1(4):045024, oct 2020.

[24] Mark H. Massé. *Rest api design rulebook*. O'Reilly, 2012.

[25] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks, 2016.

[26] Avik Pal and Aniket Das. Torchgan: A flexible framework for gan training and evaluation. *Journal of Open Source Software*, 6(66):2606, 2021.

[27] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[28] Poetry.

[29] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation, 2021.

[30] Abhinavnbsp; Ajitsaria Real Python. What is the python global interpreter lock (gil)?, May 2021.

[31] Charles Reid. Python patterns: The registry.

[32] Mark Richards and Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly, 2020.

[33] Alexander Shvets. Factory method.

[34] Alexander Shvets. Command, 2014.

[35] Ask Solem. Introduction to celery¶.

[36] Megan Stanley, John F Bronskill, Krzysztof Maziarz, Hubert Misztela, Jessica Lanini, Marwin Segler, Nadine Schneider, and Marc Brockschmidt. FS-mol: A few-shot learning dataset of molecules. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[37] GT4SD Team. GT4SD (Generative Toolkit for Scientific Discovery), 2 2022.