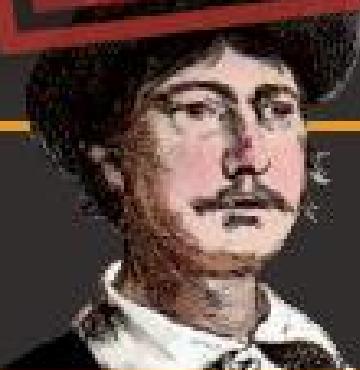


AI-Powered Developer

Build great software with ChatGPT and Copilot

Nathan B. Crocker



 MANNING

AI-Powered Developer

1. [1 Understanding Large Language Models](#)
2. [2 Getting Started with Generative AI](#)
3. [3 Designing Software with ChatGPT](#)
4. [4 Building Software with GitHub Copilot](#)
5. [5 Managing Data with GitHub Copilot and Copilot Chat](#)
6. [6 Testing, Assessing, and Explaining with Large Language Model S](#)
7. [7 Coding Infrastructure and Managing Deployments](#)
8. [welcome](#)
9. [index](#)

1 Understanding Large Language Models

This chapter covers

- Introducing Generative AI (Specifically Large Language Models)
- History of Generative AI
- Exploring the benefits of Generative AI
- Determining when and when not to use Generative AI

Whether you realized it or not, and whether you want to admit it or not, you have quietly received a promotion. In fact, every professional software engineer has. Almost overnight, we have gone from staff engineers to engineering managers. You now have the world's smartest and most talented junior developer on your team. Guiding, mentoring, and performing code reviews should become part of your daily routine. You now have Generative AI as your new coding partner. This chapter will provide you with an overview a subset of Generative AIs called Large Language Models (LLM), specifically Chat GPT, GitHub Copilot, and AWS CodeWhisperer.

Note

This book will not be a traditional programming book. You will not be able to use it like you would a script. You are going to engage with a dialogue with Large Language Models and like any conversation the words and direction will change, depending on the model, as well as the context that came before. The output that you receive will very likely differ from what is printed in this book. This should not discourage you. Instead, you should explore. The journey is as rewarding as the destination. You might find yourself frustrated that they could not follow along. Have patience. If you are disciplined (and somewhat adventurous), you can get GPT cooperate with the general themes and aim of this book: learning how to use generative AI to make you a better programmer.

1.1 An introduction to Large Language Models

Generative AIs, specifically Large Language Models (LLMs) are radically transforming how we think about and develop software. Rather than designing and coding out modules, components, and tests, we will describe the software that we want these AIs to build, and they will generate the body of this work for us. This is a natural trend in the fields of software engineering: our compilers have gotten smarter (Rust's compiler being a prime example, which eliminates an entire category of bugs), as has our tooling (IntelliSense in source code), and our programming languages have become more expressive and more productive. While this might make these Generative AIs seem more evolutionary, than a revolutionary; they are, in a sense, both.

This book will examine, compare, and contrast three such Large Language Models: GitHub's Copilot, OpenAI's ChatGPT, and Amazon's CodeWhisperer. The latter will receive the least coverage as it is largely analogous with Copilot but is more useful and focused on development within the AWS eco-system.

GitHub Copilot and ChatGPT both use the GPT-4 Codex model created by OpenAI behind the scenes. Microsoft licensed the software from OpenAI, using the source code within the public repositories within GitHub (which Microsoft owns). GitHub built a service that will take the context provided by your Integrated Development Environment (IDE), such as Visual Studio Code or IntelliJ, and send that context to the GitHub Copilot service. This Service will use the OpenAI Codex to generate up to ten possible solutions, given the context that you have provided via the comments and code in your file. Codex attempts to match this context against examples that it finds in the corpus of its training data. These code solutions will be returned to your Integrated Development Environment for you to select from. You review all of the code suggestions and accept the one that is closest to your intent. The supervision that you provide here is very important: it is not uncommon for the solutions to be out of date or inexact. Should you accept one the solutions, then your “accepted” solution is then sent back to the GitHub Copilot Service to enhance the model further and thus the suggestions.

Microsoft is betting big on this technology with Copilot. As is Google with Bard. And it is easy to see why.

This book teaches you how to manage Generative AIs such as GitHub Copilot. Trivial examples will give way to incredibly complex ones that will leave you stunned. This book assumes you have little knowledge of *using* Generative AIs. You have likely heard about Generative AIs for some time. You have probably thought the concept is exciting and worth looking into Generative AI at some point. Well, there is no time like the present. This book will take you through the basics, from setting it up in an Integrated Development Environment to using it to 10x your productivity, output, and hopefully enjoyment of coding.

Over the course of this book, you will see example after example of how one Generative AI is better suited for a given task. This will help you build intuition around when you would want to use one or the other, as well as when you might want to avoid them all. Let us take a brief survey of each of the Generative AI's core strengths.

ChatGPT excels at generating responses that mimic human speech and written language. It is therefore very good at documentation and commenting in code. Because of its ability to process Natural Languages (NLP), it can also perform the reverse: summarize text and capture the sentiment. You can also use it to improve these areas: have it rewrite or rephrase copy.

ChatGPT can generate code snippets, functions, applications, and whole chatbots. In addition, you can use it to autogenerate tests. We will do all of these things in subsequent chapters.

GitHub Copilot (and CodeWhisperer) confer the following benefits to developers: they assist with code completion, fixing errors, and refactoring. They reside within the developer's Integrated Development Environment (IDE), which can help retain a developer's focus on the task at hand. This will make developers more productive, in terms of output (lines of code per time period), but they can also automate repetitive tasks. Given that

Copilot's training data was collected by culling public repositories, the developer will have suggestions to increase the overall code quality.

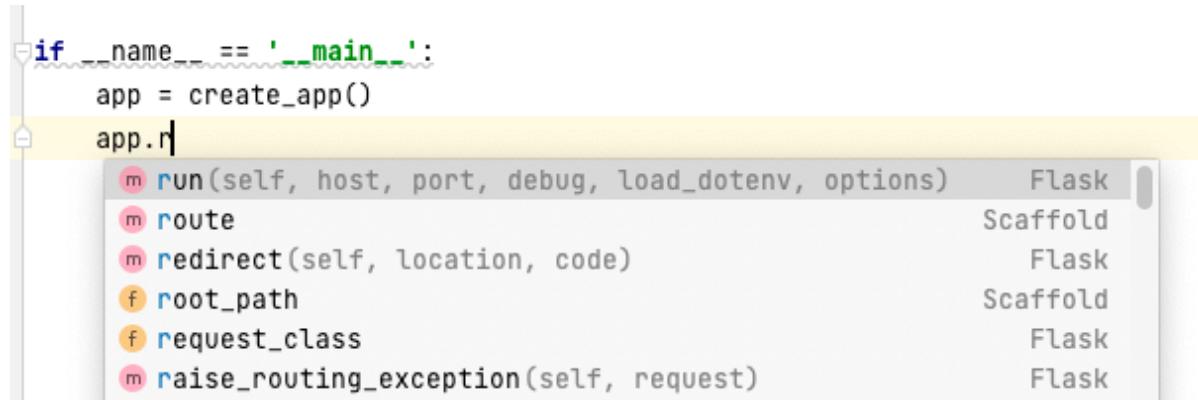
Copilot can also assist in better understanding of a foreign code base. It will provide suggestions as to how to navigate this code base, as it can help one better understand the relationships amongst the classes and code.

As you use these tools, you will notice that your velocity changes dramatically as you better understand the capabilities and limitations of your new programming partner. Your programming partner will also get better at working with you since it can “remember” your programming style and approach. Working with generative AIs will allow you to tackle much more complex problems. You will write better, cleaner code, with fewer bugs. All while moving faster than you thought possible. Sound like a dream, or worse, fluff? It isn't.

One might ask themselves Isn't this just a better version of IntelliSense? You might ask yourself this question after looking over the first, few examples; however, by the end of the next chapter, having used Generative AI, you will be able to appreciate the differences.

In figure 1.1, you will see Microsoft Visual Studio Code provide an IntelliSense auto-completion suggestion to start the Flask application. Note that this is inline, and the suggestion comes as I edit the code.

Figure 1.1 IntelliSense auto-completing the Flask run method.



The screenshot shows a code editor window in Microsoft Visual Studio Code. The code being edited is:

```
if __name__ == '__main__':
    app = create_app()
    app.r|
```

The cursor is positioned at the end of the line 'app.r|'. An auto-completion dropdown menu is open, listing several methods for the 'app' object. The methods listed are:

- m run(self, host, port, debug, load_dotenv, options) Flask
- m route Scaffold
- m redirect(self, location, code) Flask
- f root_path Scaffold
- f request_class Flask
- m raise_routing_exception(self, request) Flask

Figure 1.2 shows that GitHub Copilot has made the same suggestion based on the method name and signature. That is, it wrote the code before I started writing the implementation. The excitement behind generative AIs is tied to this power: its predictive nature. As the prompts get more explicit, the suggestions get more exact. We will explore this further in later chapters.

Figure 1.2 GitHub Copilot solution as to how to run the Flask application.



The screenshot shows a code editor interface with two tabs: 'app.py' and 'requirements.txt'. The 'app.py' tab is active and displays the following Python code:

```
# app.py 1 ●  requirements.txt
app.py > ...
1 # create a Flask application that has a route to /
2 # which returns the text "Hello World"
3 from flask import Flask
4
5 app = Flask(__name__)
6
7 @app.route('/')
8 def hello_world():
9     return 'Hello World'
10
11 if __name__ == '__main__':
12     app.run()
13
```

Figure 1.2 presents a trivial example and does not make a compelling case for why one would want to use Generative AI. However, in this same Flask application, what if you need to create an end point that can handle the input from a POST method but forget the syntax? Would we need to open the official documentation and try to find how to do it? No, we could just ask GitHub Copilot.

Figure 1.3 GitHub Copilot generating a POST method handler



The screenshot shows a code editor interface with two tabs: 'app.py' and 'requirements.txt'. The 'app.py' tab is active and displays the following Python code:

```
1 # create a Flask application that has a route to /
2 # which returns the text "Hello World"
3 from flask import Flask
4
5 app = Flask(__name__)
6
7 #define a POST method handler for the / path that returns
8 @app.route('/', methods=['POST'])
9
```

You can see that Copilot offered several similar suggestions on how to complete this code. Declaring the method would have gotten the first suggestion auto-completed inline in our IDE. No need to stop and use the mouse. This approach keeps you in the code and in the flow state for longer without unnecessary distraction. Now, if only Copilot could fetch us a coffee...

1.2 History of Generative AI

It is worth taking a quick detour to understand a little bit about the genesis of the technologies that we study throughout the next few chapters.

Generative AIs are a sub-set of Artificial Intelligence. Artificial Intelligence has been around and actively researched for more than sixty years. The Logic Theorist is considered the first application of artificial intelligence, predating the term “artificial intelligence”. The Logic Theorist was the brainchild of Herbert Simon and Allen Newell, with some contributions by Cliff Shaw. Simon and Newell were attempting to teach a computer to think.

While this attempt did not result in a genuinely thinking machine, the Logic Theorist was able to produce better, more detailed mathematical proofs than contemporary mathematicians Alfred North Whitehead and Bertrand Russell. The speculation and theorizing about what Logic Theorist would do to the field of mathematics mirrors what we are seeing in the news surrounding ChatGPT today.

The term “artificial intelligence” would not exist until the RAND Corporation hosted the Dartmouth Summer Research Project on Artificial Intelligence in 1956. Then, prominent researcher John McCarthy (the original author of Lisp) and computer scientist coined the term “artificial intelligence,” unveiling it at this conference.

Research into AI continued for decades, incrementally improving with public interest waxing and waning. Garry Kasparov was defeated by IBM’s Deep Blue beat at chess in 1997. Ken Jennings lost to IBM’s Watson at Jeopardy! in 2011. A few years later, in 2014, generative adversarial networks (GANs) were invented. The advent of GANs kicked off renewed interest in AI, as it was able to create realistic images and so-called deep fakes.

In 2017, a new type of deep learning model was invented: the transformer. This laid the groundwork for the Large Language Model (LLM): LLMs such as GPT-3. Whereas GANs could generate images from noise maps, LLMs could create text and software. Around this time, the company OpenAI – the stewards of ChatGPT – was founded.

But why was OpenAI founded? What was their goal? Let’s ask ChatGPT.

Q: what is the history of openai?

A: OpenAI is a research company that was founded in December 2015 by Elon Musk, Sam Altman, Greg Brockman, Ilya Sutskever, Wojciech Zaremba and John Schulman. The company's mission is to develop and promote friendly AI in a way that benefits humanity as a whole. OpenAI initially focused on developing and promoting AI in the gaming and robotics industries but has since expanded its focus to include a wide range of industries. In 2019, OpenAI released GPT-2, a powerful language processing AI model, and in 2020 GPT-3, which has been widely adopted in natural language processing tasks and has become one of the biggest AI models to date. In 2023, we saw the release of GPT-4, the most advanced model to date. It has the ability to connect to the internet amongst other vast improvements. OpenAI has also been involved in numerous research

projects in the fields of AI safety, robotics, and machine learning, and has established partnerships with a number of organizations and companies to advance AI research and development.

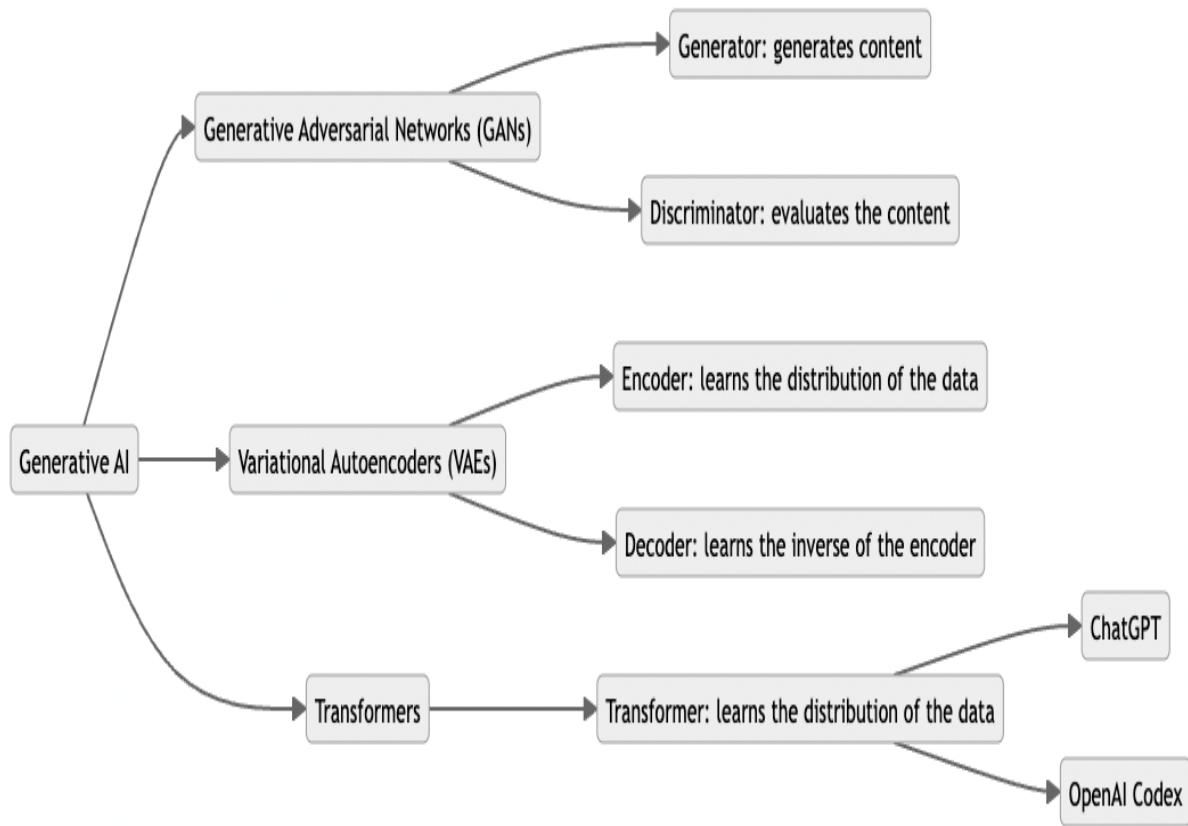
A key detail buried in this text is that there are companies that are actively working on making AI systems easier to use. Given that this trend is unlikely to end (and will likely only intensify in the future) does it not make sense to get in early? You will stay ahead of the curve. This will make your skills as an AI assisted developer all the more valuable.

1.3 Grokking Generative AI

Generative AIs are a subset of artificial intelligence. They are trained on a large dataset to learn the patterns and structures of the data. Once trained, they will use this data to generate new data similar to the dataset that they were trained on in terms of this new data's structure. This is the generative part in the name Generative AI.

There are three prominent and highly publicized types of Generative AIs: Generative Adversarial Networks (GANs), Variational Autoencoders (VAEs), and transformer-based language models. ChatGPT and OpenAI's Codex are examples of the latter. We will briefly walk through how each of these types of Generative AIs function.

Figure 1.4 There are three main types of Generative AI: Generative Models, Generative Adversarial Networks, and transformer based.

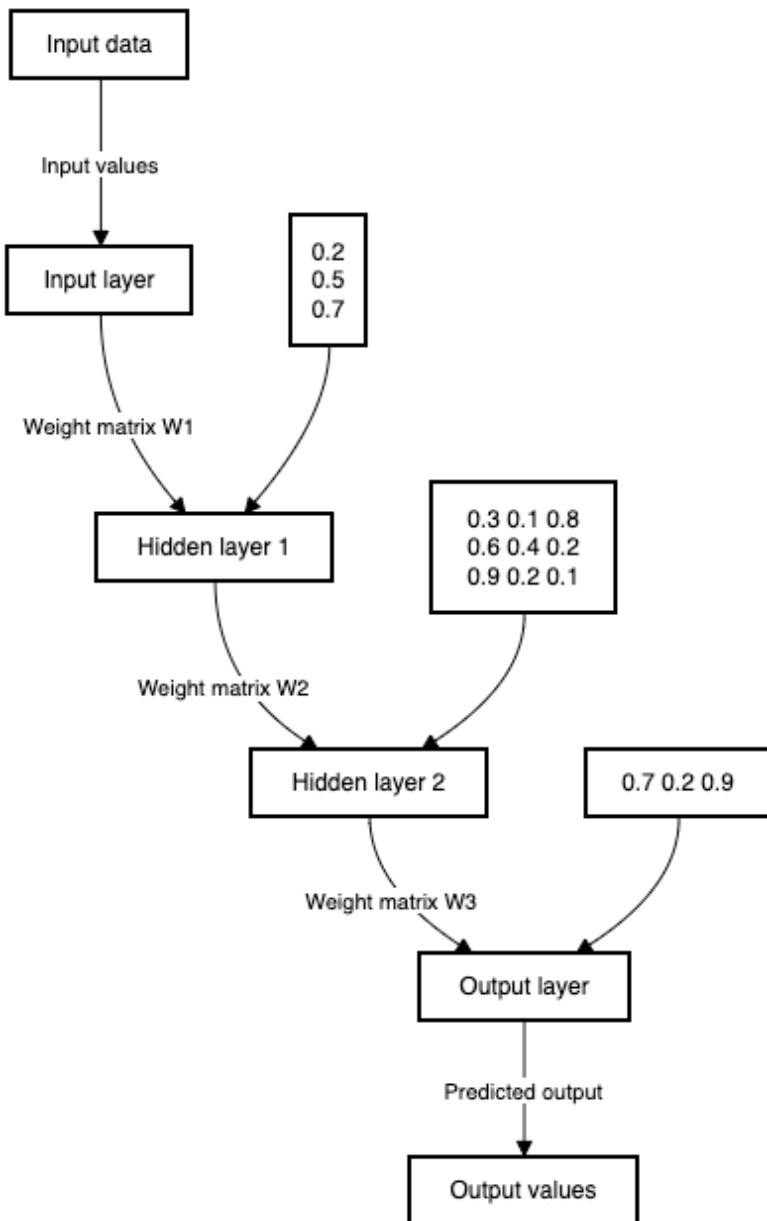


All three of these Generative AIs employ a neural network to create output; be that text or code generation or images. A neural network is patterned after how humans' minds work, as neurons pass signals to one another. You can visualize this as a directed graph in which data that exceeds certain thresholds are passed to the next node in the graph.

Data is encoding the input layer, which is called the outer layer. The output layer is connected to hidden layers. Behind the hidden layers are other, numerous hidden layers through which the data must traverse. All of the nodes in the neural network are connected via calculated numerical values, representing the strength of the connection between the neurons using back-propagation (represented as lines in Figure 1.5), which have thresholds that must be exceeded in order for the data activate the next layer. If the data makes it to the output layer, then the data is returned from the network.

There is no guarantee that data will be returned, however. The data might be filtered out.

Figure 1.5 Visualizing a neural network. A very tiny one. Outer nodes are exposed so that they can accept input. As Hidden nodes are traversed these inputs are either discarded or forwarded to the next node. If input makes to the Output nodes, then it is returned.



A Generative AI uses very large data sets to train these models. In the case of GitHub's Copilot, this large data set was the contents of the publicly accessible repositories within GitHub itself. If you have ever contributed to an open-source project, then you might have code from which Copilot has trained.

While many Generative AIs use a neural network, how they use it determines the AI's type. A Generative Adversarial Network (GAN) use two neural networks: one called generator and one called the discriminator. The generator network generates fake data based on the training data set. The discriminator tries to identify fake data. These networks are adversarial in nature as the generator attempts to create data that is indistinguishable from the real data and the discriminator attempts to discern if the data is real or fake.

Variational Autoencoders (VAEs) use two networks well: one for encoding and one for decoding. In one sense the encoding network simplifies the input by reducing the data into a lower-dimensional representation. The decoding network then maps this lower-dimensional representation back to the original data space. The whole point of this is to be able to generate new data through sampling.

The final type is *transformer-based models*. The transformer model is a type of *feedforward* neural network that uses *self-attention mechanisms* to process sequential data, such as natural language text. During training, the weights of the network are adjusted to minimize a loss function, such as cross-entropy.

In a feedforward neural network, the input flows in one direction, from input layer to output layer, with no feedback connections between the layers. Additionally, no information or error signal flows back from the output to the input layer. Therefore, the neural network's output is determined solely by the input data and the weights assigned to the connections between the layers.

Self-attention mechanisms allow the network to selectively attend to different parts of the input sequence based on their relevance to the current output. In a transformer, the input sequence is first embedded into a vector

space using an embedding layer. The embedded input sequence is then fed into an encoder network, which consists of multiple layers of feedforward neural networks. Each encoder layer applies self-attention mechanisms to capture the relationships between the different parts of the input sequence.

The self-attention mechanism calculates an attention score for each part of the input sequence based on its relationship to the other parts of the sequence. These attention scores are then used to weight the contributions of each part of the sequence to the final output of the encoder network. This allows the encoder network to selectively focus on the most important parts of the input sequence, while ignoring irrelevant parts.

The output of the encoder network is then fed into a decoder network, which also consists of multiple layers of feedforward neural networks. The decoder uses self-attention mechanisms to generate an output sequence based on the input sequence, one token at a time.

An analogy for the relationship between encoders and decoders in a transformer network is that of a compiler and linker. Just as a compiler breaks down high-level code into low-level instructions and a linker combines those instructions into a final executable program, the encoder network breaks down the input sequence into meaningful units and the decoder network combines those units into a final output sequence. The use of self-attention mechanisms in transformers is similar to the way a compiler and linker optimize code for better performance.

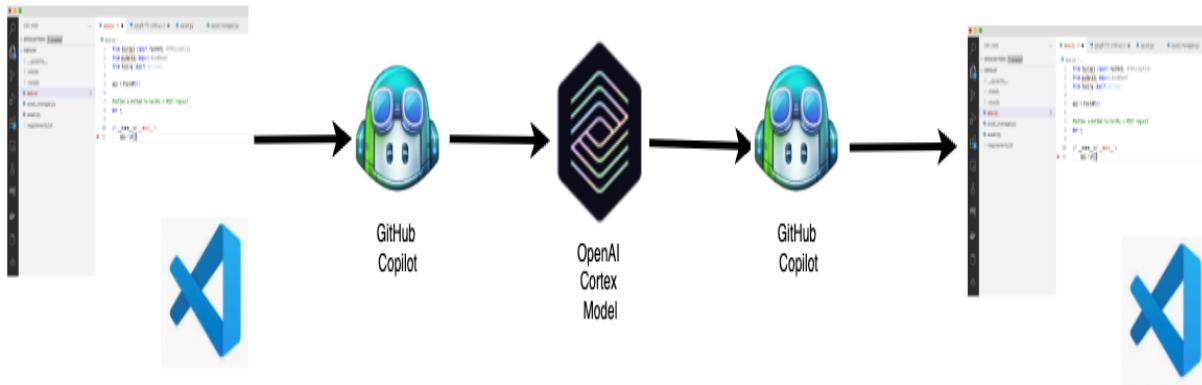
As previously stated, many Generative AIs use neural networks, but not all of them do. Some are rules-based, generating output by applying the rules to the inputs. Still others are evolutionary in nature, iterating on results, and selecting based on the goodness of fit.

We can now walk through a descriptive example of how you would interact with Copilot. As you begin to type in your favorite IDE (VS Code in this example), the Copilot plugin will send your comments or code (sometime all it takes is a function's name!) into the Copilot service.

This service turns your lines of code or comments into natural language prompts, which are then run through OpenAI's Codex model. The model will generate suggestions based on the training data set. GitHub refers to this as code synthesis. They claim that this training data set contains billions of lines of code from dozens of programming languages.

Once the top ten solutions are returned from the Codex model, the Copilot service will return these suggestions back to your editor. You select the suggestion that most accurately captures your intent or need. Or you reject all of the suggestions. Your selection is then returned to the Copilot service to better train the model.

Figure 1.6 Your code is sampled and encoded by the Copilot plug-in. It is then set to the OpenAI Codex model, where suggestions are generated. These suggestions are then returned to your VS Code session.



GitHub is constantly improving their Copilot service. A recent release (as of December 2022), they boast that the acceptance rate is 46% across all programming languages and with Java specifically, it is 61% on average^[1].

1.4 When to use and when to avoid Generative AI

Excitement around these technologies is growing. Since its public release in late November of 2022, there are hundreds (possibly thousands) of articles

about the various dimensions of ChatGPT. Will it ruin education? Is education required anymore? Are software engineers necessary anymore?

It is easy to give into the pessimism. There are lots of unknowns and the full impact of this technology has yet to be revealed. However, you should form your own opinion as you work through this book. It is my hope that you will see the positives of generative AI and use them for good. You will use it to grow as a programmer. As you use them, you will become a better programmer.

One of the best ways to grow as a developer is to read good code. OpenAI has curated some of the best code on the planet; it is all at your fingertips. You also now have some of the worst code available at your fingertips as well. You can learn from good examples as well as bad. Being able to discern the difference is the key to growing.

So, when should you use generative AIs? Every opportunity that you can! (We will discuss some exceptions to their usage shortly.) It is fascinating to engage with generative AIs. You will learn how to use them better, find shortcuts, discover new features, and have starry-eyed child-like delight every moment of it.

While employing generative AIs in your daily job would appear to make a lot of sense (because it does), it should be noted that it is not pertinent to use in all cases.

If you were given a take home coding exam, should you use a generative AI to complete this exam? Unless it was explicitly stated that you can, then you should avoid it. It could be construed as cheating if the tester did not anticipate you using it.

Should they anticipate that you would use you? Yes, they probably should at this point. Further, one might argue that, given that the purpose of the exam is to assess the coding abilities of potential candidates, potential employers should try to construct real world conditions in order to best assess one's ability. This should include all tools and resources that one would have

available to them doing the course of their day. These tools would include generative AIs.

You should be especially careful of using them in an academic setting. Proper attribution is complex, at a minimum, and the line between inspiration and plagiarism with Generative AI is a very fine one. If your usage is determined to be plagiarism, the consequences will be dire and permanent: expulsion and you might be barred from future enrollment at other institutions. Proceed with extreme caution.

In general, use your better judgement. If there is a chance that you might run afoul of any copyright laws or administrative policy, then do not use it unless you are granted specific permission to do so. It would not hurt to talk to your corporate IT or InfoSec team at work, just to make certain that you comply with corporate policies related to corporate systems and computers.

One final note: Generative AIs are tools and like any tool, you need to have some idea of what you are doing. You should have some idea about what the correct answer should be. You should use them in domains, in which you have some idea what is going on. In cases like these, you will find yourself exploring the landscape, deepening your understanding of the domain as well as learning faster.

Now that we have explored an abridged history of Generative AI, seen some use cases for Generative AI, and applied some important guardrails, we will go hands-on in the next chapter, examining how to start the same project using these three Generative tools.

1.5 Summary

- Generative AIs are both evolutionary and revolutionary. Evolutionary in the sense that they are just another iterating on the tools that we as developers use every day. Revolutionary in that they will transform how we do our jobs. In fact, they will change our jobs.
- The future of development will be the management of Generative AI. Even the mythical 10x developer will not have the productivity of a

developer with an AI partner; an AI powered developer will produce higher quality code at a substantially faster rate, at lower cost than one who is not. We will spend more of our time training our AI partner to do what we want, how we want, then we will write code without the AI.

- While there are several Generative AI out in the wild, we will explore three of the most popular:
 - ChatGPT – Has been making headlines since November 2022.
 - GitHub Copilot – The most popular Generative AI that uses in an Integrated Development Environment. Financed and promoted by Microsoft.
 - Amazon Web Services CodeWhisperer – A product similar to Copilot but backed by Amazon.
- Some of the world's biggest companies are making a significant investment in Generative AIs (Microsoft, Amazon, Alphabet) and making them easier to use (Open AI).
- Generative AIs make use of extremely sophisticated neural networks, resembling ours, to filter and map input to new previously unseen output.
- You should check with your professor or teacher prior to making use of generative AIs for school work.

[1] Shuyin Zhao. “GitHub Copilot now has a better AI model and new capabilities.” *GitHub Blog*. <https://github.blog/2023-02-14-github-copilot-now-has-a-better-ai-model-and-new-capabilities/>. Last accessed: Feb 14, 2023.

2 Getting Started with Generative AI

This chapter covers

- Engaging with ChatGPT
- Learning the basics of using Copilot
- Learning the basics of using CodeWhisperer
- Contrasting the differences between these three Generative AI offerings

In this chapter, we will dive into using three of the significant Generative AIs AI tools: ChatGPT, GitHub Copilot, and AWS CodeWhisperer. The primary focus of this chapter will be to employ these AI tools in the development of a model Information Technology Asset Management (ITAM) system, specifically concentrating on the hardware management component.

ITAM systems, both commercial and open source, offer a range of functionalities dedicated to an organization's software and hardware management. By restricting the scope of our project to the hardware management component alone, we aim to highlight how we interact with each Generative AI tool, compare the quality of the code they generate, and contrast their distinct features and limitations.

An ITAM system, on the surface, appears relatively straightforward, primarily focusing on asset management, tagging, and tracking. However, as we shall uncover, there's a significant layer of complexity hidden within these seemingly simple requirements. This inherent complexity makes our chosen project an ideal candidate for employing a Generative AI to navigate the problem-space.

So, let's embark on this fascinating journey of building and understanding. We'll begin with an overview of the system we aim to construct and the roles that these remarkable AI tools will play in this process.

2.1 Introducing our project, the Information Technology Asset Management system

An *Information Technology Asset Management* (ITAM) system is a tool to manage and track hardware devices, software licenses, and other IT-related components throughout their lifecycle. ITAM systems typically consist of hardware and software inventory tools, license management software, and other related software applications. The system may also involve manual processes and physical tracking of IT assets using QR codes, barcodes, or other physical asset management technologies.

Generally, ITAM systems will have a centralized database, which stores the asset identifiers and attributes specific to the asset type. For example, you might store device type, model number, operating system, and installed applications for desktop PCs. For software, you might hold the application's name, the vendor, the number of licenses available, and on which computers the software has been installed. The latter ensures that your organization complies with all licensing restrictions. By monitoring usage, you should never exceed the number of licenses you have purchased.

ITAM systems also confer the ability to control costs. Since you always know what software and hardware you have available, you should not have to make any unnecessary purchases. These systems centralized purchases, which can help with volume purchasing. Unused hardware can be sold. Underutilized hardware can have their workloads consolidated. Additionally, as we will see, we can use this purchase date information to calculate the depreciation value of the hardware, applying this value against your organization's taxes.

We will explore more of the features of an ITAM system, building a miniature version, one focused on hardware management. We will begin with ChatGPT, building out the first component using Python. We will then replicate this exercise using Copilot and AWS CodeWhisperer. By repeating this process with each of these Generative AI tools, we can contrast the differences and correlate the similarities.

First, starting with ChatGPT, we will focus on building out the Asset class. Next, we will develop a means to present it via a representational state transfer (REST) controller using the library FastAPI. Finally, we will build on this in subsequent chapters, adding features, extending the design, and adding documentation.

Why Python?

In 2023, one should not need to justify the decision to use Python 3 in one's application development. However, it is worth enumerating why choosing Python for this book makes sense, given the topic. Firstly, Python is an excellent language for prototyping projects. Data scientists and machine learning specialists use it frequently for this reason. Secondly, it is expressive and productive. A few lines can get a good deal done, especially given the size of the standard library, not to mention the commonly used libraries such as pandas and numpy. Thirdly, it is easy to learn and use. Its syntax focuses on (and enforces) readability. Finally, it is supported by all of the Generative AI platforms.

2.2 A Foray into ChatGPT

Context is one of the most important aspects of working with ChatGPT. Your previous *prompts* can drastically change the results from your current prompt. In the context of language models like ChatGPT, a prompt refers to the input provided to the model to generate a response. It can be a single sentence, a paragraph, or even a longer piece of text. It serves as the instruction or query to the model, guiding its response. Given the quality of the prompt forms the context in which the model responds, it is important to always be aware of what prompts you have issued in the current session. Therefore, starting with a new session every time you begin a new project is advised. Appendix A will walk you through setting up an account, logging into ChatGPT, and writing your first prompt.

We will start with the simple prompt, "Using Python, please write an Asset class for an ITAM." It bears repeating when working with Large Language

Models the same input will not always produce the same output. For this reason, your output might not match the following exactly; hopefully, however, it resembles it.

Listing 2.1 The Asset class generated by ChatGPT when asked to create a class for an ITAM

```
class Asset:
    def __init__(self, asset_id, asset_type, manufacturer,
model, purchase_date, warranty_end_date, location):
        self.asset_id = asset_id
        self.asset_type = asset_type
        self.manufacturer = manufacturer
        self.model = model
        self.purchase_date = purchase_date
        self.warranty_end_date = warranty_end_date
        self.location = location
        self.assigned_to = None
        self.status = 'Available'

    def assign_to_user(self, user):
        self.assigned_to = user
        self.status = 'Assigned'

    def unassign(self):
        self.assigned_to = None
        self.status = 'Available'

    def get_details(self):
        details = f "Asset ID: {self.asset_id}\n"
        details += f"Type: {self.asset_type}\n"
        details += f"Manufacturer: {self.manufacturer}\n"
        details += f"Model: {self.model}\n"
        details += f "Purchase Date: {self.purchase_date}\n"
        details += f "Warranty End Date:
{self.warranty_end_date}\n"
        details += f"Location: {self.location}\n"
        details += f "Assigned To: {self.assigned_to}\n"
        details += f"Status: {self.status}\n"
        return details
```

In addition to the code, ChatGPT will return documentation for this class.

Figure 2.1 ChatGPT enumerates the attributes in the Asset class as well as the class's methods

This implementation of the Asset class has the following attributes and methods:

- Attributes:
 - `asset_id`: a unique identifier for the asset
 - `asset_type`: the type of the asset, e.g. laptop, desktop, server, printer, etc.
 - `manufacturer`: the manufacturer of the asset
 - `model`: the model of the asset
 - `purchase_date`: the date the asset was purchased
 - `warranty_end_date`: the date the asset's warranty ends
 - `location`: the current location of the asset
 - `assigned_to`: the user who the asset is currently assigned to (if any)
 - `status`: the current status of the asset (available or assigned)
- Methods:
 - `assign_to_user(user)`: assigns the asset to a user, changing the status to "Assigned"
 - `unassign()`: unassigns the asset from its current user, changing the status to "Available"
 - `get_details()`: returns a string with the details of the asset, including all attributes and their values.

Note that this implementation is a basic example and can be extended as per the specific requirements of your ITAM system.

To track our Assets, we will need a way to interact with them. An API is an excellent way to interact with objects in your system. REST is one way to expose your API functionality in a microservice architecture. Therefore, we will ask ChatGPT to create a REST controller for this class with the following prompt, "Please create a REST controller for this class." The abridged ChatGPT generated code is in listing 2.2.

Listing 2.2 The REST controller generated by ChatGPT when asked to add a REST controller to the project

```
from flask import Flask, request, jsonify
from asset import Asset, AssetManager

app = Flask(__name__)
asset_manager = AssetManager()

# Get all assets
@app.route('/assets', methods=['GET'])
def get_assets():
    assets = [asset.__dict__ for asset in asset_manager.assets]
    return jsonify({'assets': assets})

# Get an asset by ID
@app.route('/assets/<string:asset_id>', methods=['GET'])
def get_asset(asset_id):
    asset = asset_manager.get_asset_by_id(asset_id)
    if asset:
        return jsonify(asset.__dict__)
    else:
        return jsonify({'error': 'Asset not found'})

if __name__ == '__main__':
    app.run(debug=True)
```

ChatGPT made some interesting assumptions. Firstly, it used Flask to build out the REST controller. While Flask is a popular, Python based web framework and by no means a bad decision, it would be of interest to us why it was selected. So, we can ask ChatGPT why it chose Flask. Use the following prompt, "Why did you use Flask for this controller?" Part of the enjoyment of working with Large Language Models is to attempt to peel back the proverbial layers to see how it works.

Figure 2.2 ChatGPT explains the rationale for using Flask when it was not provided a framework to use. It states that it is lightweight and easy to use.



I used Flask for this REST API controller because it is a lightweight and easy-to-use web framework for Python, especially for building simple REST APIs. Flask provides a simple and flexible way to handle HTTP requests and responses, and it doesn't require a lot of boilerplate code to get started. It also has a large community and a lot of resources and tutorials available online, making it easy to learn and use.



In addition, Flask is a good choice for small to medium-sized projects, as it doesn't require a lot of setup or configuration, and it can be easily extended with a variety of third-party plugins and libraries.

Overall, Flask provides a good balance between simplicity and flexibility, making it a good choice for building a simple REST API controller for the `Asset` class.

While the rationale is good, it is not the framework we wanted to use. Shortly, we will have ChatGPT switch the application to use FastAPI. First, however, we should ensure that the code we just “wrote” will run. However, we are missing some dependencies in our project. We can ask ChatGPT to create a requirements.txt file for us.

Enter the following prompt in the input field, "Please create a requirements.txt file for this project." ChatGPT should respond with something similar to the following code snippet.

Listing 2.3 The requirements.txt file for the Asset management project with Flask support

```
click==8.0.3
Flask==2.1.1
importlib-metadata==4.8.1
itsdangerous==2.0.1
Jinja2==3.0.2
MarkupSafe==2.0.1
typing-extensions==3.10.0.2
Werkzeug==2.0.1
zipp==3.6.0
```

Once we create and populate the initial app.py file (the initial Python code that ChatGPT created) and the requirements.txt file we just generated, we will notice the second assumption that ChatGPT made: the AssetManager class. ChatGPT silently generated that code without cluing us in. This hidden class for which we have no source code means our project will not run.

Thankfully, we can ask ChatGPT to give us this code. Enter the following prompt, "Please show me the source code to AssetManager."

Listing 2.4 The class definition for the AssetManager class generated by ChatGPT

```
class AssetManager:
    def __init__(self):
        self.assets = []

    def add_asset(self, asset):
        self.assets.append(asset)

    def remove_asset(self, asset_id):
        for asset in self.assets:
            if asset.asset_id == asset_id:
                self.assets.remove(asset)
```

```

        return True
    return False

def get_asset_by_id(self, asset_id):
    for asset in self.assets:
        if asset.asset_id == asset_id:
            return asset
    return None

def assign_asset(self, asset_id, user):
    asset = self.get_asset_by_id(asset_id)
    if asset and asset.status == 'Available':
        asset.assign_to_user(user)
        return True
    return False

def unassign_asset(self, asset_id):
    asset = self.get_asset_by_id(asset_id)
    if asset and asset.status == 'Assigned':
        asset.unassign()
        return True
    return False

def get_available_assets(self):
    return [asset for asset in self.assets if asset.status
== 'Available']

def get_assigned_assets(self):
    return [asset for asset in self.assets if asset.status
== 'Assigned']

```

Additionally, ChatGPT will produce documentation listing the class usage, as well as all methods of the class. This documentation would be helpful if we were looking how to use the class, or if we wanted to add comments to our code.

Figure 2.3 ChatGPT provides documentation on the AssetManager class, including the attributes and methods.

The `AssetManager` class provides methods for managing a collection of `Asset` objects. Here's what each method does:

- `__init__`: Initializes an empty list of assets.
- `add_asset`: Adds an `Asset` object to the list of assets.
- `remove_asset`: Removes an `Asset` object from the list of assets by its `asset_id`.
- `get_asset_by_id`: Returns an `Asset` object from the list of assets by its `asset_id`.
- `assign_asset`: Assigns an `Asset` object to a user by its `asset_id`.
- `unassign_asset`: Unassigns an `Asset` object from its user by its `asset_id`.
- `get_available_assets`: Returns a list of all `Asset` objects that are available (not assigned to a user).
- `get_assigned_assets`: Returns a list of all `Asset` objects that are currently assigned to a user.

These methods provide the basic functionality needed to manage a collection of IT assets in an ITAM system.

Should you choose to store the class AssetManager in a separate Python file, which is a good idea, then you will need to update the app.py file, changing the import statements this way:

Listing 2.5 Fixing the import statements of the file app.py

```
from flask import Flask, request, jsonify
from asset import Asset
from asset_manager import AssetManager

app = Flask(__name__)
asset_manager = AssetManager()
...
```

You should be able to run our very simple ITAM system now. Once you start the Flask application, if you navigate to <http://localhost:5000/assets>, you should see Assets as JSON output.

Figure 2.4 We added three Assets to the AssetManager. The controller returned this collection in JSON format



```
{
  "assets": [
    {
      "asset_id": "001",
      "asset_type": "Laptop",
      "assigned_to": null,
      "location": "New York",
      "manufacturer": "Dell",
      "model": "Latitude E7470",
      "purchase_date": "2021-01-01",
      "status": "Available",
      "warranty_end_date": "2024-01-01"
    }
  ]
}
```

There is one outstanding issue to which we should turn our attention. This project is still using Flask. There are several reasons why we prefer to use FastAPI instead of Flask. The first of which is that FastAPI would give us

better performance. FastAPI consistently ranks significantly better in benchmarks, thus the name. FastAPI has support for asynchronous non-blocking, highly concurrent programming. Asynchronous non-blocking systems yield better performance on the same hardware because the computer will execute other tasks while it is waiting for I/O operations to complete. However, the main reason that we want to use FastAPI in this project is to illustrate how easy it is to change our middleware with ChatGPT.

Therefore, we should switch to FastAPI. We can ask ChatGPT to do this for us. Enter the following prompt, "Please change this project from using Flask to using FastAPI." ChatGPT will generate code similar to the following app.py. Note that listing 2.6 is abridged, including only the GET REST operations. The full output would have support for POST, PUT, and DELETE.

Listing 2.6 The updated app.py which includes FastAPI support

```
app = FastAPI()
asset_manager = AssetManager()

class AssetIn(BaseModel):
    asset_id: str
    asset_type: str
    manufacturer: str
    model: str
    purchase_date: str
    warranty_end_date: str
    location: str

class AssetOut(BaseModel):
    asset_id: str
    asset_type: str
    manufacturer: str
    model: str
    purchase_date: str
    warranty_end_date: str
    location: str
    assigned_to: str
    status: str
```

```
@app.get("/assets/{asset_id}", response_model=AssetOut)
async def get_asset(asset_id: str):
    asset = asset_manager.get_asset_by_id(asset_id)
    if asset:
        return asset
    else:
        return {"message": "Asset not found"}
```

There are several things to note. The first is that ChatGPT informs us of the changes it made to the application. See figure 2.5. Next, notice that it created two additionally embedded classes: AssetIn and AssetOut. This is considered good practice, as it provides an interface or contract the clients should adhere to, while allowing us the freedom to change our internal representation of our domain model.

Figure 2.5 ChatGPT explains the changes that it made to the application to change it from a Flask application to FastAPI.

Here's a brief description of the changes made to the file:

- `from flask import Flask` was replaced with `from fastapi import FastAPI`.
- `@app.route` decorators were replaced with `@app.post`, `@app.delete`, `@app.get`, and `@app.put` decorators with appropriate HTTP methods.
- JSON input and output validation was added using Pydantic data models, which were defined using the `BaseModel` class.
- The response models for the `get_asset`, `get_available_assets`, and `get_assigned_assets` endpoints were specified using the `response_model` parameter.
- The `request.json` property was replaced with the corresponding request and response models for each endpoint.

Note that while this code implements the same functionality as the original Flask code, there may be differences in how FastAPI handles certain aspects of the REST API, such as routing, error handling, and performance.

When changing the middleware of a project, you should check to see if the domain model classes have not changed. In this case, both Asset and AssetManager have changed. We should ask ChatGPT to show us those new classes.

Note

As you design and develop software with ChatGPT, it is important to periodically review the code for different classes or files. This is because the context is continuously built up through your interactions with ChatGPT. This review process will allow you to see firsthand how these changes have been implemented.

First, let us examine the changes that have been made to the Asset class. To do this, enter the following prompt: "Please show the code for Asset."

Listing 2.7 The updated Asset class reflecting the change in middleware from Flask to FastAPI

```
from typing import Optional

class Asset:
    def __init__(self, asset_id: str, asset_type: str,
manufacturer: str, model: str, purchase_date: str,
warranty_end_date: str, location: str):
        self.asset_id = asset_id
        self.asset_type = asset_type
        self.manufacturer = manufacturer
        self.model = model
        self.purchase_date = purchase_date
        self.warranty_end_date = warranty_end_date
        self.location = location
        self.assigned_to: Optional[str] = None
        self.status: str = 'Available'

    def assign_to_user(self, user: str):
        self.assigned_to = user
        self.status = 'Assigned'

    def unassign(self):
```

```
self.assigned_to = None
self.status = 'Available'

def get_details(self):
    details = f "Asset ID: {self.asset_id}\n"
    details += f"Type: {self.asset_type}\n"
    details += f"Manufacturer: {self.manufacturer}\n"
    details += f"Model: {self.model}\n"
    details += f "Purchase Date: {self.purchase_date}\n"
    details += f "Warranty End Date:
{self.warranty_end_date}\n"
    details += f"Location: {self.location}\n"
    details += f "Assigned To: {self.assigned_to}\n"
    details += f"Status: {self.status}\n"
    return details
```

The most meaningful change that ChatGPT added is the optional type hint. This would signal your intent that this attribute – assigned_to – should not be Nil and therefore cause a NoneType error. As usual, ChatGPT also helpfully included method documentation.

Figure 2.6 ChatGPT explains the updated Asset class. While it does not expressly tell us the changes that it enacted when the class was updated, it does provide fulsome documentation on the class's methods.

The `Asset` class provides methods for managing an individual IT asset. Here's what each method does:

- `__init__`: Initializes a new `Asset` object with the specified asset ID, type, manufacturer, model, purchase date, warranty end date, and location. The object is initially set to be available and not assigned to a user.
- `assign_to_user`: Assigns the asset to the specified user and sets its status to "Assigned".
- `unassign`: Unassigns the asset from its current user and sets its status to "Available".
- `get_details`: Returns a formatted string with the details of the asset, including its ID, type, manufacturer, model, purchase date, warranty end date, location, assigned user (if any), and status.

This class provides the basic functionality needed to manage individual IT assets in an ITAM system using Python.

Next, enter the following prompt: "Please show the code for AssetManager ." The constructor initializes a List of Assets. This List is the data structure on which the AssetManager class will use to perform the CRUD (Create, Read, Update, and Delete) operations.

Listing 2.8 The updated AssetManager class reflecting the change in middleware from Flask to FastAPI

```
from typing import List, Optional
from asset import Asset

class AssetManager:
    def __init__(self):
        self.assets: List[Asset] = []

    def add_asset(self, asset: Asset):
        self.assets.append(asset)
```

The remove_asset method will iterate over the elements in the List of Assets. If it finds the Asset with the identifier supplied in the method's parameter (asset_id), it will remove it from the List.

Listing 2.9 The remove_asset method of the AssetManager

```
def remove_asset(self, asset_id: str) -> bool:
    for asset in self.assets:
        if asset.asset_id == asset_id:
            self.assets.remove(asset)
            return True
    return False
```

The get_asset method iterates over all elements in the List, looking for the Asset with the asset_id passed in as a parameter. If it finds this Asset then it will return it. This search functionality is very similar to many of the methods in this class: you will notice this again in the assign_asset and unassign_asset methods. If we were to observe the DRY (Don't Repeat Yourself) principle (and we should!), then we would refactor this class (or ask ChatGPT to do it for us), consolidating the logic by exacting a single method for use throughout the class.

Listing 2.10 The common methods of the AssetManager class

```
def get_asset_by_id(self, asset_id: str) -> Optional[Asset]:
    for asset in self.assets:
        if asset.asset_id == asset_id:
            return asset
    return None

def assign_asset(self, asset_id: str, user: str) -> bool:
    for asset in self.assets:
        if asset.asset_id == asset_id:
            if asset.status == 'Available':
                asset.assign_to_user(user)
                return True
            else:
                return False
    return False
```

```
def unassign_asset(self, asset_id: str) -> bool:
    for asset in self.assets:
        if asset.asset_id == asset_id:
            if asset.status == 'Assigned':
                asset.unassign()
                return True
            else:
                return False
    return False
```

Similarly, the `get_available_asset` and `get_assigned_asset` methods contain common functionality. Both find all Assets whose status matches the parameter. These would be candidates for refactoring the code into a common methods that each uses.

Listing 2.11 Methods to return Assets with a given status

```
def get_available_assets(self) -> List[Asset]:
    return [asset for asset in self.assets if asset.status
== 'Available']

def get_assigned_assets(self) -> List[Asset]:
    return [asset for asset in self.assets if asset.status
== 'Assigned']

def get_all_assets(self) -> List[Asset]:
    return self.assets
```

Accompanying this code was documentation related to the class's methods and their usage.

Figure 2.7 ChatGPT lists the methods that are available in the AssetManager class, now that it has been migrated to use FastAPI

The `AssetManager` class provides methods for managing a collection of `Asset` objects. Here's what each method does:

- `__init__`: Initializes a new `AssetManager` object with an empty list of assets.
- `add_asset`: Adds the specified `Asset` object to the asset list.
- `remove_asset`: Removes the `Asset` object with the specified ID from the asset list, if it exists.
- `get_asset_by_id`: Returns the `Asset` object with the specified ID from the asset list, if it exists.
- `assign_asset`: Assigns the `Asset` object with the specified ID to the specified user, if it is available.
- `unassign_asset`: Unassigns the `Asset` object with the specified ID from its current user, if it is assigned.
- `get_available_assets`: Returns a list of all available `Asset` objects in the asset list.
- `get_assigned_assets`: Returns a list of all assigned `Asset` objects in the asset list.
- `get_all_assets`: Returns a list of all `Asset` objects in the asset list.

This class provides the basic functionality needed to manage a collection of IT assets in an ITAM system. The implementation of the `AssetManager` class may vary depending on the requirements of the ITAM system, but this implementation should give you an idea of how to manage a collection of assets using Python.

Finally, we will need ChatGPT to regenerate the dependencies file as well. Use the following prompt, "Please show me the new requirements.txt file."

Listing 2.12 The updated requirements.txt file, which uses FastAPI

```
fastapi==0.70.0
uvicorn==0.15.0
pydantic==1.8.2
```

Once you have re-run the pip command (`pip install -r requirements.txt`) to install the updated dependencies, you should be able to run the project using the following command: `uvicorn app:app --reload`.

The port has now changed to 8000. When I ran this, there wasn't a method to display all Assets in the AssetManager. We can ask ChatGPT to add such a method. "Please add a method to `app.py` to return and display all Assets." You will likely get an asynchronous method that returns all of the Assets for the AssetManger. The code should resemble the following listing.

Listing 2.13 The method used to display all of the Assets in the AssetManager

```
@app.get("/assets/", response_model=List[AssetOut])
async def get_assets():
    assets = asset_manager.get_all_assets()
    return assets
```

Routing in FastAPI

In this method, we define a new route using the `@app.get` decorator, with a URL path of `/assets/`. We also specify a response model of `List[AssetOut]` using the `response_model` parameter, which tells FastAPI to validate the output data and generate API documentation based on the schema of the `AssetOut` data model.

Inside the method, we call the `get_all_assets` method of the `asset_manager` object, which returns a list of all `Asset` objects. We then return this list to the client, which will be automatically converted to a JSON array by FastAPI.

You can test this endpoint using a web browser or a tool such as curl. For example, if you are running the application locally on port 8000, you could open `http://localhost:8000/assets/` in a web browser to see a JSON array of all assets.

Let's test our ITAM system by adding and displaying a new Asset in our browser. You can use curl or Postman to add a new Asset.

Listing 2.14 Using curl to add new Asset to the AssetManager

```
curl -X POST -H "Content-Type: application/json" -d
'{"asset_id": "12345", "asset_type": "Laptop", "manufacturer": "Dell", "model": "Latitude 5400", "purchase_date": "2022-01-01", "warranty_end_date": "2023-01-01", "location": "New York"}'
http://localhost:8000/assets/
```

You will likely get an error when you attempt to buy this new Asset. Two modifications are required to get the code that ChatGPT generated to work. The first is to add a new method to the Asset class called `to_dict()`. FastAPI uses the library pydantic to convert the In/Out models (e.g. `AssetIn` and `AssetOut`) to and from the Asset class. It does this by using a dictionary to assign the attributes. The `to_dict()` method will give us an easy way to provide that dictionary.

Listing 2.15 Add the `to_dict` method to the Asset class defined in asset.py

```
def to_dict(self):
    return {
        "asset_id": self.asset_id,
        "asset_type": self.asset_type,
        "manufacturer": self.manufacturer,
        "model": self.model,
        "purchase_date": self.purchase_date,
        "warranty_end_date": self.warranty_end_date,
        "location": self.location,
        "assigned_to": self.assigned_to,
        "status": self.status
    }
```

The second change we need to do is to translate the output model (`Assetout`) to JSON. We would do that everywhere that we output this object. We will examine the changes in the class that relate to changing the output to JSON.

Listing 2.16 An updated app.py class with JSON encoding of output

```
@app.get("/assets/", response_model=List[AssetOut])
async def get_assets():
    assets = asset_manager.get_all_assets()
    return JSONResponse(content=[asset.to_dict() for asset in assets])

@app.post("/assets/")
async def add_asset(asset: AssetIn):
    new_asset = Asset(asset.asset_id, asset.asset_type,
asset.manufacturer, asset.model, asset.purchase_date,
asset.warranty_end_date, asset.location)
    asset_manager.add_asset(new_asset)
    return {"message": "Asset added successfully"}

@app.delete("/assets/{asset_id}")
async def remove_asset(asset_id: str):
    if asset_manager.remove_asset(asset_id):
        return {"message": "Asset removed successfully"}
    else:
        return {"message": "Asset not found"}

@app.get("/assets/{asset_id}", response_model=AssetOut)
async def get_asset(asset_id: str):
    asset = asset_manager.get_asset_by_id(asset_id)
    if asset:
        return JSONResponse(content=asset.to_dict())
    else:
        return {"message": "Asset not found"}

@app.put("/assets/{asset_id}/assign")
async def assign_asset(asset_id: str, user: str):
    if asset_manager.assign_asset(asset_id, user):
        return {"message": "Asset assigned successfully"}
    else:
        return {"message": "Asset not available"}

@app.put("/assets/{asset_id}/unassign")
async def unassign_asset(asset_id: str):
    if asset_manager.unassign_asset(asset_id):
        return {"message": "Asset unassigned successfully"}
    else:
        return {"message": "Asset not assigned"}

@app.get("/assets/available/", response_model=List[AssetOut])
```

```

async def get_available_assets():
    assets = asset_manager.get_available_assets()
    return JSONResponse(content=[asset.to_dict() for asset in assets])

@app.get("/assets/assigned/", response_model=List[AssetOut])
async def get_assigned_assets():
    assets = asset_manager.get_assigned_assets()
    return JSONResponse(content=[asset.to_dict() for asset in assets])

```

Switching the middleware from Flask to FastAPI caused a myriad of issues in our code. However, going through this process helped us build an intuition as to how and when we should engage with ChatGPT. ChatGPT is an indispensable tool to aid in our software design arsenal. It can allow us to quickly make radical changes to our projects and give us a sense of the direction we should take. It tends to fail (at least as of this writing) at very fine tasks.

One interesting experiment that you should try (in a new session) create a prompt to create this project using FastAPI first and compare that against the project that we created iteratively. You should notice that the code is drastically different.

In this section of the book, we delved into the practical application of ChatGPT in software development, by developing a small IT Asset Management (ITAM) project. We harnessed ChatGPT to automatically generate essential elements of the software, including creating classes, developing a RESTful controller, and generate a requirements.txt file. The process underscored the capabilities of ChatGPT as an assistive tool in expediting and simplifying the complex processes of software development, particularly for smaller scale projects such as this one. The application of ChatGPT in this context not only highlighted its utility in building fundamental software components, but also demonstrated the potential it holds for streamlining software development processes in future.

2.3 Let Copilot take control!

Let us now use GitHub Copilot to tackle the creation of the Asset class. Appendix B has instructions on how to create an account and install the plugin into your favorite IDE (assuming your favorite IDE is either VS Code or PyCharm). Once you have completed the installation you should create a new project in your favorite IDE. First, create a new file named asset.py. This class will hold the source code for the Asset class. As you type the constructor, Copilot will make code suggestions. You will likely get code resembling the source code in figure 2.9. Alternatively, you could activate interactive model by pressing `Ctrl+Enter`, and Copilot will generate up to ten suggestions based off of the current context.

NOTE

Throughout the book we will mostly use inline mode, in which you use the `Tab` key to accept a given suggestion.

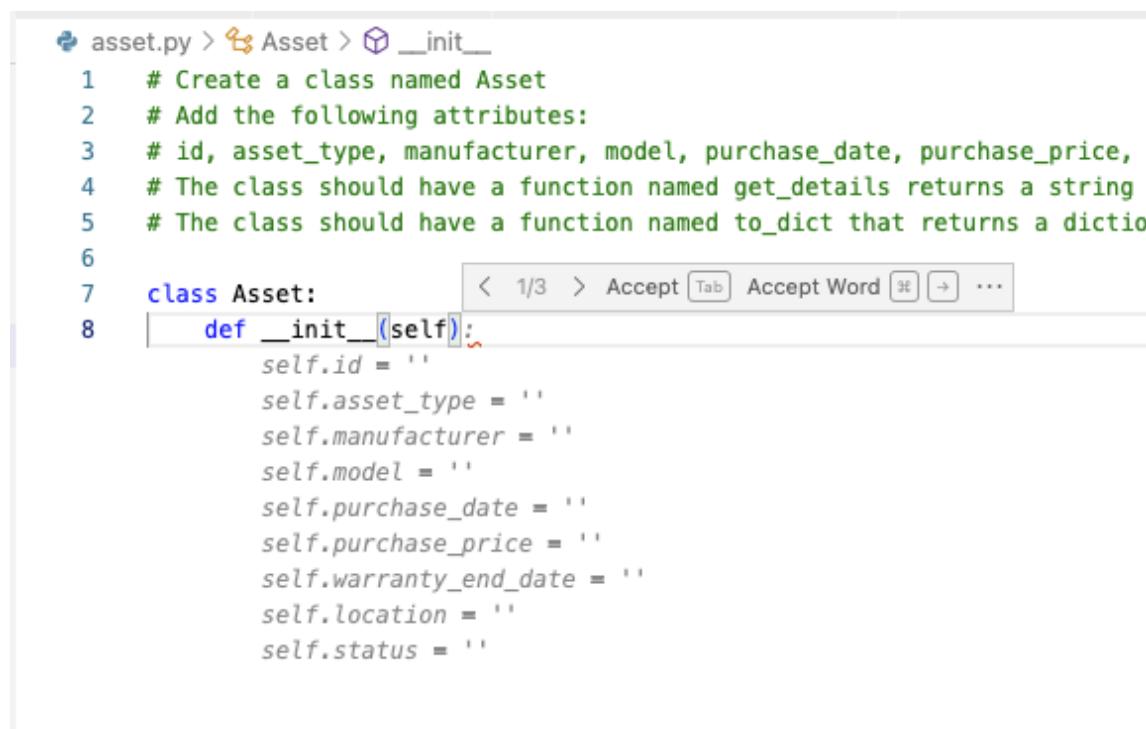
While Copilot's first attempt was admirable, it does not resemble the class definition we were interested in; we want an ITAM system asset, not a generic implementation of an asset. Like what we did with ChatGPT, we will create a prompt that will signal our intention to Copilot, giving it additional context and information that it can use to generate a code suggestion. Entering prompts in Copilot is as simple as adding a comment at the beginning of the source code file. We will tell Copilot what the name of the class should be and what attributes it should have.

Listing 2.17 The starting prompt for GitHub Copilot asks it to create a new class named Asset

```
# Create a class named Asset
# Add the following attributes: id, asset_type, manufacturer,
model, purchase_date, purchase_price, warranty_end_date,
location, and status.
# The class should have a function named get_details that
returns a string that is the concatenation of all attributes.
# The class should have a function named to_dict that returns a
dictionary of all attributes.
```

As you begin to define the constructor, Copilot will return an entire code suggestion for an ITAM system asset. If we made a side-by-side comparison of the ChatGPT-generated code with the resulting code from Copilot, we would notice that they closely resemble one another.

Figure 2.9 Based off of the comment that we added to top of the file, Copilot was able to generate a constructor, containing all of the attributes included in the prompt.



The screenshot shows a code editor with a file named 'asset.py'. The code defines a class 'Asset' with an __init__ method. The code is as follows:

```
# Create a class named Asset
# Add the following attributes:
# id, asset_type, manufacturer, model, purchase_date, purchase_price, warranty_end_date, location, status
# The class should have a function named get_details returns a string
# The class should have a function named to_dict that returns a dictionary
class Asset:
    def __init__(self):
        self.id = ''
        self.asset_type = ''
        self.manufacturer = ''
        self.model = ''
        self.purchase_date = ''
        self.purchase_price = ''
        self.warranty_end_date = ''
        self.location = ''
        self.status = ''
```

The code editor interface includes tabs for 'Accept', 'Accept Word', and '...', and a status bar indicating '1/3'.

Next, add the methods `get_details` and `to_dict`. The `JSONResponse` class accepts a dictionary and convert it to JSON for us. The `to_dict` method will help facilitate this handoff. The `get_details` method would be useful during debugging, enabling us to see the contents of the class.

Listing 2.18 The code that Copilot generates in response to our request

```
# Create a class named Asset
# Add the following attributes: id, asset_type, manufacturer,
model, purchase_date, purchase_price, warranty_end_date,
location, and status.
```

```

# The class should have a function named get_details that
# returns a string that is the concatenation of all attributes.
# The class should have a function named to_dict that returns a
# dictionary of all attributes.

class Asset:
    def __init__(self):
        self.id = None
        self.asset_type = None
        self.manufacturer = None
        self.model = None
        self.purchase_date = None
        self.purchase_price = None
        self.warranty_end_date = None
        self.location = None
        self.status = None

    def get_details(self):
        return f"{self.id}, {self.asset_type},\n{self.manufacturer}, {self.model}, {self.purchase_date},\n{self.purchase_price}, {self.warranty_end_date},\n{self.location}, {self.status}"

    def to_dict(self):
        return {
            "id": self.id,
            "asset_type": self.asset_type,
            "manufacturer": self.manufacturer,
            "model": self.model,
            "purchase_date": self.purchase_date,
            "purchase_price": self.purchase_price,
            "warranty_end_date": self.warranty_end_date,
            "location": self.location,
            "status": self.status
        }

```

Now that we have the asset class. We will move on to the AssetManager class. We will start with a basic prompt, telling Copilot to create the AssetManager class and define some basic functionality.

Listing 2.19 The prompt to have Copilot generate our AssetManager

```
# Create a class called AssetManager that will allow us to store  
and retrieve assets.
```

Shortly after defining the class, you will notice that Copilot will suggest the complete source code for this manager.

Figure 2.10 With minimal instruction Copilot was able to create an AssetManager that allowed for the storage and retrieval of Assets.

```
# Create a class called AssetManager that will allow us to store and retrieve assets.  
  
class AssetManager:  
    def __init__(self):  
        self.assets = []  
  
    def add_asset(self, asset):  
        self.assets.append(asset)  
  
    def get_asset(self, id):  
        for asset in self.assets:  
            if asset.id == id:  
                return asset  
        return None  
  
    def get_assets(self):  
        return self.assets  
  
    def update_asset(self, id, asset):  
        for i, a in enumerate(self.assets):  
            if a.id == id:  
                self.assets[i] = asset  
                return asset  
        return None  
  
    def delete_asset(self, id):  
        for i, a in enumerate(self.assets):  
            if a.id == id:  
                self.assets.pop(i)  
                return True  
        return False
```

Copilot's source code for the AssetManager will closely resemble the code that ChatGPT generated.

Listing 2.20 The complete AssetManager source code generated by Copilot from our initial prompt

```
# Create a class called AssetManager that will allow us to store
and retrieve assets.

class AssetManager:
    def __init__(self):
        self.assets = []

    def add_asset(self, asset):
        self.assets.append(asset)

    def get_assets(self):
        return self.assets

    def get_asset(self, asset_id):
        for asset in self.assets:
            if asset.id == asset_id:
                return asset
        return None

    def update_asset(self, asset_id, asset):
        for index, asset in enumerate(self.assets):
            if asset.id == asset_id:
                self.assets[index] = asset
                return True
        return False

    def delete_asset(self, asset_id):
        for index, asset in enumerate(self.assets):
            if asset.id == asset_id:
                del self.assets[index]
                return True
        return False
```

Next, we will assemble the FastAPI application to run and display the Assets. We will begin with the following prompt at the beginning of the app.py class.

Listing 2.21 The starting prompt to have Copilot build our FastAPI application

```
# Create a FastAPI app that allows CRUD operations on the Asset class.
```

Figure 2.11 Copilot begins to automagically import the required libraries.

```
app.py
1 # Create a FastAPI app that allows CRUD operations on the Asset class.
2
3 from fastapi import FastAPI, HTTPException
4
5
```

Copilot will add the import statements. Something interesting happens after the definition of the FastAPI application: Copilot will attempt to define the Asset class. It is good practice to encapsulate and protect your internal model, which is the Asset class.

Figure 2.12 Copilot attempts to define a pydantic-based Asset class. However, we have already defined an Asset class, which represents our "internal" model of the class. We, therefore, rejected this suggestion.

```
app.py > ...
1 # Create a FastAPI app that allows CRUD operations on the Asset class.
2
3 from fastapi import FastAPI, HTTPException
4 from pydantic import BaseModel
5 from typing import Optional
6
7 app = FastAPI()
8
9 class Asset(BaseModel):
10
```

To protect our internal model, we will, therefore, reject the suggestion that ChatGPT provided us. Instead, we will create two classes: AssetIn and AssetOut. The Copilot-generated code closely mimics the solution that

ChatGPT supplied to us. Once we type in AssetIn, Copilot will helpfully create the rest of the class for us, as it has intuited what we want to do.

Figure 2.13 Copilot has determined that we do not want to directly expose our internal model and create a class representing the input payload from our yet-to-be-defined POST REST endpoint.

```
app.py > AssetIn
1  # Create a FastAPI app that allows CRUD operations on the Asset class.
2
3  from fastapi import FastAPI, HTTPException
4  from pydantic import BaseModel
5  from typing import Optional
6
7  app = FastAPI()
8
9  class AssetIn(BaseModel):
10     asset_type: str
11     manufacturer: str
12     model: str
13     purchase_date: str
14     purchase_price: float
15     warranty_end_date: str
16     location: str
17     status: str
```

As we continue to define the methods in our FastAPI application, Copilot will predict the methods' implementation.

Figure 2.14 The final contents of the file `app.py` after Copilot has generated the implementation of all of the methods.

```

app = FastAPI()
asset_manager = AssetManager()

class AssetIn(BaseModel):
    asset_id: str
    asset_type: str
    manufacturer: str
    model: str
    purchase_date: str
    warranty_end_date: str
    location: str

class AssetOut(BaseModel):
    asset_id: str
    asset_type: str
    manufacturer: str
    model: str
    purchase_date: str
    warranty_end_date: str
    location: str
    assigned_to: Optional[str]
    status: str

@app.get("/assets/", response_model=List[AssetOut])
async def get_assets():
    assets = asset_manager.get_assets()
    return JSONResponse(content=[asset.to_dict() for asset in assets])

```

Finally, we will get Copilot to create a requirements.txt file for us. We begin with the prompt in listing 2.22. Then, Copilot will add the version numbers as we start enumerating the libraries.

Listing 2.22 The requirements.txt file for this project

```

# generate a requirements.txt file for this project
fastapi==0.63.0
uvicorn==0.13.4
pydantic==1.7.3
starlette==0.13.6

```

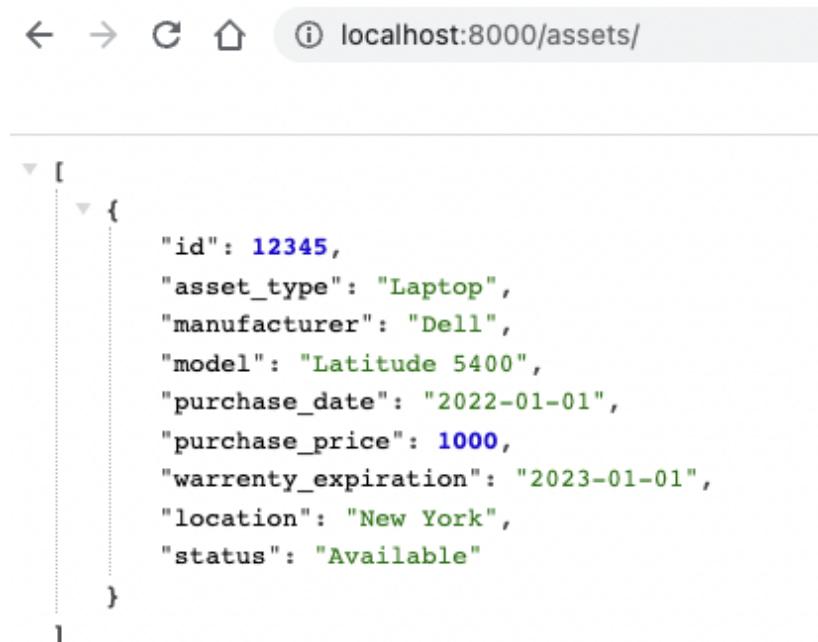
Start the application using the following command: uvicorn app:app --reload. Create a new Asset will the curl command in listing 2.23.

Listing 2.23 The curl command to add a new Asset to our Copilot application

```
curl -X POST -H "Content-Type: application/json" -d '{"id": "12345", "asset_type": "Laptop", "manufacturer": "Dell", "model": "Latitude 5400", "purchase_date": "2022-01-01", "warrenty_expiration": "2023-01-01", "location": "New York", "status": "Available", "purchase_price": "1000.00"}'  
http://localhost:8000/assets/
```

Open your browser to <http://localhost:8000/assets/> to display a new asset via the REST controller.

Figure 2.15 We created a single Asset object using the POST endpoint. This is displayed using the GET endpoint.



The screenshot shows a browser window with the address bar containing 'localhost:8000/assets/'. The main content area displays a JSON object representing an asset:

```
[{"id": 12345, "asset_type": "Laptop", "manufacturer": "Dell", "model": "Latitude 5400", "purchase_date": "2022-01-01", "purchase_price": 1000, "warrenty_expiration": "2023-01-01", "location": "New York", "status": "Available"}]
```

In this section, we've embarked on a comparative journey of software development using two distinct Generative AI tools - ChatGPT and GitHub Copilot, both applied to the same IT Asset Management (ITAM) project. ChatGPT provided significant flexibility in our development approach, even enabling us to switch frameworks, albeit with some caution. It served as a valuable advisor, proffering suggestions and guidance.

Meanwhile, our experience with Copilot required a different approach. We used prompts to instruct Copilot to generate code, which required us to have a clear vision of the intended implementation from the outset.

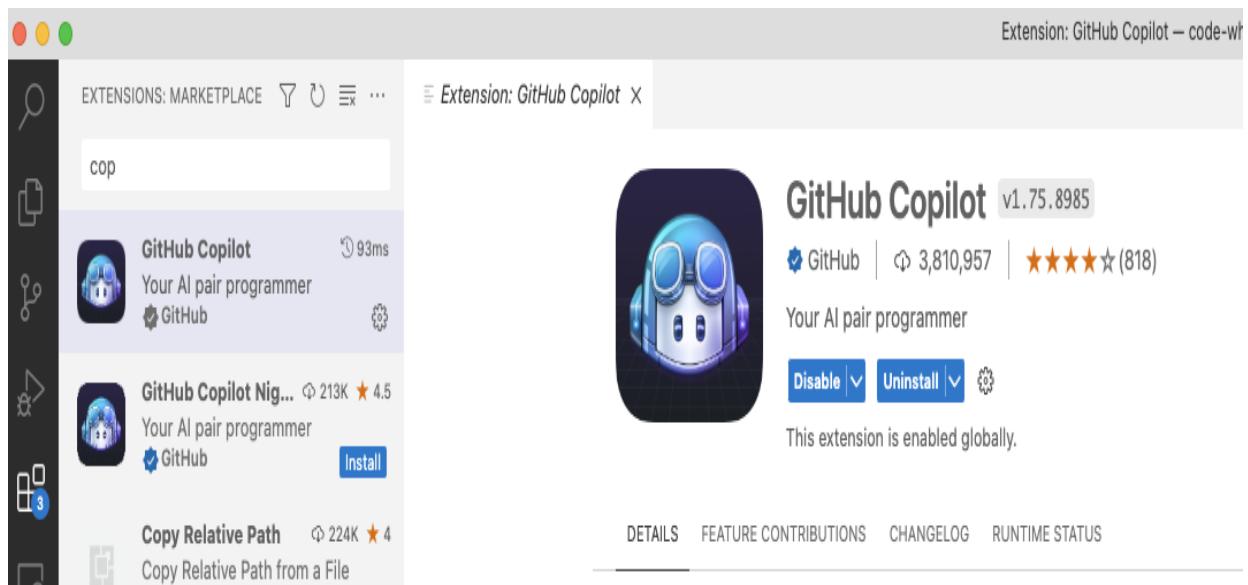
Next, we plan to introduce AWS CodeWhisperer to the mix, adding another layer to this explorative exercise. Finally, we will draw comparisons across all three tools - ChatGPT, Copilot, and AWS CodeWhisperer. The objective remains to understand their unique strengths and limitations, and how each tool can optimally be used depending on the requirements and contexts of software development projects.

In conclusion, through our hands-on experience with these AI-powered tools, we've delved into the transformative potential that AI holds for the realm of software development. Whether it's ChatGPT, Copilot, or the upcoming foray into AWS CodeWhisperer, each of these tools brings something unique to the table, promising a more efficient and streamlined software development process. However, the key to maximizing their benefits lies in our ability to continually adapt and learn, making the most of each tool while being cognizant of their limitations.

2.4 Let CodeWhisperer Speak Loudly

Before we begin using CodeWhisperer, we should disable Copilot. Click on the Extension tab and search for Copilot. Once you have found it click the **Disable** button. You will need to restart the application. Once the application has restarted, you can begin to use CodeWhisperer. If you find yourself in need of assistance to install or configure CodeWhisperer, you should refer to Appendix C. Once the plugin is installed and you are logged into your developer account, you should create a file called asset.py.

Figure 2.16 Before we can switch to CodeWhisperer, we must disable Copilot. We disable the extension in the Extensions tab.



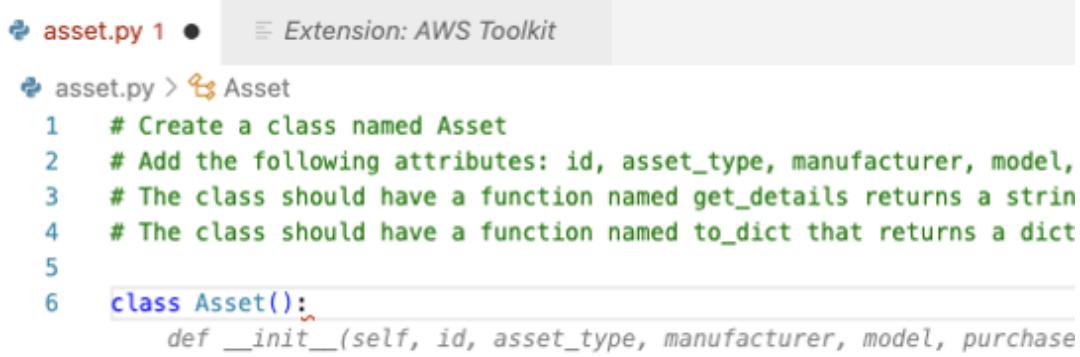
We will use the same prompt that we used for Copilot. It is reprinted here for convenience.

Listing 2.24 The prompt to have CodeWhisperer create the Asset class for us

```
# Create a class named Asset
# Add the following attributes: id, asset_type, manufacturer,
model, purchase_date, purchase_price, warranty_end_date,
location, and status.
# The class should have a function named get_details that
returns a string that is the concatenation of all attributes.
# The class should have a function named to_dict that returns a
dictionary of all attributes.
```

Once we have entered this prompt and written the class definition, CodeWhisperer will begin writing the constructor.

Figure 2.17 CodeWhisperer suggests a constructor based on the prompt we entered at the beginning of the file.



The screenshot shows a code editor window titled "asset.py 1". The status bar indicates "Extension: AWS Toolkit". The code itself is as follows:

```
# Create a class named Asset
# Add the following attributes: id, asset_type, manufacturer, model,
# The class should have a function named get_details returns a strin
# The class should have a function named to_dict that returns a dict
#
class Asset():
    def __init__(self, id, asset_type, manufacturer, model, purchase
```

We then continue to write the method definitions, and CodeWhisperer will dutifully generate the implementation.

Listing 2.25 The complete code generated by CodeWhisperer

```
# Create a class named Asset
# Add the following attributes: id, asset_type, manufacturer,
# model, purchase_date, purchase_price, warranty_end_date,
# location, and status.
# The class should have a function named get_details that
# returns a string which is the concatenation of all attributes.
# The class should have a function named to_dict that returns a
# dictionary of all attributes.

class Asset():
    def __init__(self, id, asset_type, manufacturer, model,
    purchase_date, purchase_price, warranty_end_date, location,
    status):
        self.id = id
        self.asset_type = asset_type
        self.manufacturer = manufacturer
        self.model = model
        self.purchase_date = purchase_date
        self.purchase_price = purchase_price
        self.warranty_end_date = warranty_end_date
        self.location = location
        self.status = status

    def get_details(self):
        return f"{self.id}, {self.asset_type},
{self.manufacturer}, {self.model}, {self.purchase_date},
{self.purchase_price}, {self.warranty_end_date},
```

```
{self.location}, {self.status}"  
  
    def to_dict(self):  
        return {"id": self.id, "asset_type": self.asset_type,  
"manufacturer": self.manufacturer, "model": self.model,  
"purchase_date": self.purchase_date, "purchase_price":  
self.purchase_price, "warranty_end_date":  
self.warranty_end_date, "location": self.location, "status":  
self.status}
```

The final code should resemble the code created by Copilot. Extremely similar. Given this, we will skip writing the other files in the project. Completing the project will be left to a motivated reader.

In this section, we introduced AWS CodeWhisperer into the mix. Like its predecessors, CodeWhisperer capably generated key components of the project, reinforcing the transformative potential of AI in the realm of software development.

Given the striking similarity of the code produced by each of these tools, an intriguing question naturally arises: how do these products truly compare? The answer to this question is not as straightforward as one might think, given the unique strengths and limitations that each tool possesses.

In the following section, we'll delve into this very question, comparing these three tools - ChatGPT, Copilot, and AWS CodeWhisperer - in a bid to understand their unique offerings, optimal use cases, and how they might reshape the future of software development. We aim to provide a comprehensive guide that can help software developers navigate this rapidly evolving landscape of AI-driven tools.

2.5 Comparing ChatGPT, Copilot, and CodeWhisperer

The first dimension we will consider is the engagement model: how we engage with AI. In the case of ChatGPT, we log into the chat website and enter prompts into a chat input box. Then, we refine our requirements in subsequent prompts. The feedback loop takes the context from the previous

prompts, applies it to the current prompt, and generates output, to which the user reacts and refires. If we contrast this engagement model against the model of Copilot and CodeWhisperer, we note that the latter two tools work within an Integrated Development Environment. We do not use it outside our IDE; try as we might. The approach is not inherently inferior; it just differs.

The way that Copilot and CodeWhisperer keep you in your IDE could be seen as a benefit rather than a deficiency. In the forthcoming Copilot X release, you will get the best of both worlds: ChatGPT and GPT 4.0 all within your IDE. These tools keep you in your code without distraction for longer. Working distraction-free is one of the keys to productivity. Copilot and CodeWhisperer excel at getting out of your way; keeping you from switching contexts; keeping you free from distraction; keeping you in the flow state longer. They do this well. You engage ChatGPT in a dialog; Copilot and CodeWhisperer advise you. The dialog will take longer to work through; advice comes fast and free.

Next, we examine how the code is presented and generated. ChatGPT can create the code as a block, method, class, or project. ChatGPT reveals projects deliberatively if asked. But ChatGPT did create the project behind the scenes. ChatGPT, after all, likes to talk. With Copilot and CodeWhisperer, the code unfolds one method at a time, at least initially. As you use it more, you will notice that they can write more and more of the code for a given class. They will, however, and unfortunately, not be able to write an entire project with a tiny prompt.

One item that they all share is their ability to respond to prompts. With ChatGPT, prompts are the only way to engage with the tool. With Copilot and CodeWhisperer, responding to prompts is not strictly necessary, but coding such prompts will make the output correspond more closely to what you initially had in mind.

Combining these factors, one might conclude that ChatGPT is an excellent choice for exploration and prototyping. However, ChatGPT can introduce unnecessary distractions, partly because you have left your IDE and are now in a Web Browser with all of the accompanying temptations that come with a browser. ChatGPT, itself, is part of the inclusion of unnecessary distractions.

You will eventually fall down the proverbial rabbit hole. The tool makes it too easy, not too. Don't let that scare you off. It is a wonderful resource.

Copilot and CodeWhisperer require that you have a desired outcome in mind. Therefore, these tools are perfect for when you want to head down, coding with precise requirements and tight deadlines. Copilot and CodeWhisperer work best when you know the language and the framework. They can automate much of the drudgery, allowing you to focus on the business requirements, which add value and are likely why you are writing the software in the first place.

The following chart briefly summarizes these benefits and limitations of all three AIs:

Figure 2.18 A comparison of the positives and negatives of ChatGPT, Copilot, and CodeWhisperer.

The diagram illustrates the support levels of three tools (ChatGPT, Copilot, and CodeWhisperer) across various development tasks. The tools are represented by icons: ChatGPT by a diamond, Copilot by a triangle, and CodeWhisperer by a triangle. The tasks are listed in rows, and the support levels are indicated by the presence or absence of these icons.

	ChatGPT	Copilot	CodeWhisperer
Prompt-based	◊	△	△
IDE-based	○	◊	◊
Generate methods	△	△	△
Generate classes	△	△	△
Generate projects	△	○	○
Generate documentation	△	△	△
Switch languages	△	○	○
Switch libraries	△	△	△

Legend:

- ◊ Exclusively
- △ Supports
- Unsupported

We went through a lot in this chapter, revisiting the same code several times. However, writing three versions of the same subsection of the ITAM system was foundational, allowing us to recognize when it would make sense to use ChatGPT as opposed to when to use the other IDE-focused tools such as Copilot and CodeWhisperer. In subsequent chapters, we will leverage this knowledge, choosing the most suitable tool.

One final note, these tools work best when they work together. ChatGPT is an excellent tool for example and structure. Copilot and CodeWhisperer provide you with the ability to extend and customize the code. This makes the forthcoming Copilot X release so exciting: it is a marriage of both of these approaches.

2.6 Summary

- An Information Technology Asset Management (ITAM) system is a tool to manage and track hardware devices, software licenses, and other IT-related components throughout their lifecycle. These systems allow companies to control costs, manage vendor relationships, and ensure licensing and regulatory compliance.
- In this chapter, we explored using ChatGPT, GitHub Copilot, and AWS CodeWhisperer by writing the same code in all three tools. Writing the same code thrice allowed us to compare the features and contrast the differences in usage: how and when to use each.
- ChatGPT is a prompt-based Generative AI that engages the user in a dialog that helps them explore ideas to aid in the design and development of entire projects. In addition, ChatGPT artfully generates documentation for each method that it writes. One of the reasons that we began the chapter using it is because it helped define a template we used throughout the remainder of the chapter. It is a fascinating product, one that can lead to unnecessary albeit enjoyable distractions.
- Copilot and CodeWhisperer are heads-down tools, working best when you know what you want to do and need some advice on how best to get it done. How one engages with these tools is remarkably similar, as are the results.
- ChatGPT (as of this writing) does not support development within an IDE. However, unlike GitHub Copilot and AWS CodeWhisperer, it can produce entire projects and easily translate code from one programming language to another. GitHub Copilot and AWS CodeWhisperer take hints from your comments to infer what code you want to be written. With ChatGPT, you explicitly write prompts that ChatGPT will use to create the code.

3 Designing Software with ChatGPT

This chapter covers

- Prototyping potential designs with ChatGPT
- Documenting our architecture in Mermaid
- Completing our design with ChatGPT

Now that we have an intuition about when to use Generative AI, we will start to design, explore, and document our application's architecture. Laying out some of the critical components upfront is beneficial in several ways. For example, it allows us to delegate some of the design to sub-architects or some of the development to other team members. Designing up front will also help us clarify our thinking about the implementation, allowing us to anticipate and avoid some pitfalls. Finally, capturing the design as documentation enables us to justify our crucial design decisions, communicating our intent to our future selves, our stakeholders, and those who may inherit the project.

First, let's solicit ChatGPT's opinion on how best to design this project and see what solution it can come up with.

3.1 Petitioning ChatGPT to aid us in our system design

In a new session, we will begin with a prompt, outlining our requirements. We should be able to construct an elaborate prompt given that we spent much of the last chapter thinking about our requirements and should have an excellent idea about what is needed. Alternatively, we could petition ChatGPT for the requirements for such a system. We could then incorporate these requirements into our prompt, editing as needed.

Undoubtedly, you have encountered endless articles purporting to demonstrate the correct way to do prompt engineering. Prompt engineering,

or the practice of designing and optimizing the initial input or "prompt" given to the model to obtain the desired output., is an integral part of how we engage with Large Language Models. Much of what we do within this book would be considered prompt chaining, generated knowledge prompting, and zero-shot inference. These are primarily academic ways of saying that we will be engaging in a dialog with our generative AI tools. The important takeaway is, as with any conversation, when you need a specific answer, you ask a particular question. Or you request that the generative AI think through the problem step by step, and be as specific as possible. For this reason, we will provide ChatGPT with a very specific prompt, to formalize the requirements gathering process.

Listing 3.1 A prompt featuring the requirements for our system

Please take the following criteria for an information technology asset management system and create a Python project that satisfies them:

Asset Inventory: The system should be able to maintain a complete inventory of all hardware and software assets owned by an organization.

Asset Tracking: The system should be able to track asset location, status, and usage. This includes information such as who uses the asset, when it was last serviced, and when it is due for replacement.

Asset Lifecycle Management: The system should be able to manage the entire lifecycle of an asset, from procurement to disposal. This includes tracking warranty and lease information, managing asset maintenance and repair, and ensuring compliance with regulations.

Asset Allocation: The system should be able to allocate assets to employees or departments, track usage, and manage reservations for assets that are in high demand.

Asset Reporting: The system should be able to generate reports on asset utilization, availability, and maintenance history. Reports should be customizable and easy to understand.

Integration: The system should be able to integrate with other IT systems, such as procurement, help desk, and financial systems, to ensure consistency and accuracy of data. **Security:** The system should have robust security features to protect sensitive information about assets, including user authentication, role-based access control, and data encryption.

Scalability: The system should be scalable to accommodate changes in the organization's IT infrastructure, including new hardware and software assets, as well as changes in the number of users and departments.

User-Friendly Interface: The system should have a user-friendly interface that is easy to navigate, search, and use. It should also be customizable to meet the needs of individual users and departments.

Accessibility: The system should be accessible from different devices and locations, including mobile devices and remote areas. It should also be accessible to users with disabilities.

After some deliberation, ChatGPT will return the high-level steps of what is needed to be done to develop this project. Oh, who are we kidding, it produces these steps immediately. These suggestions are helpful, but we are here to get past the empty file problem: we want to start designing. We want prose and diagrams to help us understand the system at the component level. We need tasks not a recipe. We should ask ChatGPT to show its work.

Listing 3.2 A prompt to have ChatGPT to show its work

Please show the package structure

At this prompt, ChatGPT will generate and return a package diagram (this listing is slightly abridged), demonstrating that not only can it design software, but it can aid in the layout of our physical files. We might quibble as to the overall package structure (or we may not), but this will provide us with more than enough inspiration to get going.

Listing 3.3 The package structure for the Asset Management System that ChatGPT created

```
asset_management_system/
└── app/
    ├── __init__.py
    ├── api/
    │   ├── __init__.py
    │   ├── asset.py
    │   ├── user.py
    │   └── ...
    ├── auth/
    │   ├── __init__.py
    │   ├── login.py
    │   ├── register.py
    │   └── ...
    ├── models/
    │   ├── __init__.py
    │   ├── asset.py
    │   ├── user.py
    │   └── ...
    ├── views/
    │   ├── __init__.py
    │   ├── asset.py
    │   ├── user.py
    │   └── ...
└── requirements.txt
└── run.py
```

We can examine the macro, and we can ask ChatGPT to provide us with the micro as well. We could create prompts to have ChatGPT show us to contents of every file in sequence. We could dutifully copy the contents down to a file with the same name as the once suggested by ChatGPT. We would always want to double check ChatGPTs work, however, we should never assume that it is 100% safe or accurate. We will occasionally find that ChatGPT does not produce code that works. ChatGPT is focused on generating the next token, rather than thinking in systems as we do. That being said, we should have a quick look at the code that it generated to assess the quality and make a determination as to its usefulness.

Listing 3.4 A prompt to have ChatGPT to show its (detailed) work

```
please show the source code for
asset_management_system/models/asset.py
```

ChatGPT will return the source code for this class. It uses Flask and SQLAlchemy, so the domain model is a little muddled. Next time, we should ask it to use our preferred application server and keep the domain model clean. These are additional data points for us as well. Now we have an idea of what we don't want.

Listing 3.5 The example source code for the Asset model in asset_management_system/models/asset.py:

```
from datetime import datetime
from asset_management_system import db

class Asset(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable=False)
    description = db.Column(db.String(255))
    category = db.Column(db.String(80))
    purchase_date = db.Column(db.DateTime,
default=datetime.utcnow)
    purchase_cost = db.Column(db.Float)
    warranty_expiry = db.Column(db.Date)
    lease_expiry = db.Column(db.Date)
    status = db.Column(db.String(80), default="Available")
    location = db.Column(db.String(80), default="IT Department")
    assigned_to = db.Column(db.String(80))
    assigned_date = db.Column(db.DateTime)
    retired_date = db.Column(db.DateTime)
    created_date = db.Column(db.DateTime,
default=datetime.utcnow)
    updated_date = db.Column(db.DateTime,
default=datetime.utcnow, onupdate=datetime.utcnow)

    def __repr__(self):
        return '<Asset %r>' % self.name
```

Now that we have a sense for how ChatGPT would tackle the problem, let's get a second opinion. To do this we will explore existing projects and libraries. We will also use ChatGPT to do this investigation.

3.2 Surveying the Landscape

If we have never used or seen an Information Technology Asset Management (ITAM) system before, we would want to understand what it is, what it should do, and how to implement such a system. As we experienced in the last chapter, this is a task to which ChatGPT is well suited. GitHub Copilot and CodeWhisperer are excellent tools when the problem space is known, when you are ready for head-down highly productive coding. ChatGPT, on the other hand, can be used to explore this problem space, to prototype interactively, and to design and document your solution. A good first step in exploring the problem space is to examine current open-source projects. We can use these "out of the box" projects or leverage their design to inspire ours. Using ChatGPT, we can quickly navigate the open-source world and zero in on any projects that might be analogous to what we are attempting to build. We can start with the prompt in listing 3.1.

Listing 3.6 A prompt to find out what ITAM projects are out there

What are some examples of ITAM projects or products?

ChatGPT responds with several examples, summarizing their core features. However, since we are looking for inspiration for our development project, we should start to refine this list to only those for which the code is available; that is, which are open-source projects?

Figure 3.1 ChatGPT presents a list of attributes of the features of an ITAM and several products and projects that possess those features.

ITAM (IT Asset Management) is the process of managing the lifecycle of an organization's IT assets, from procurement to disposal. There are several projects and products that can help with ITAM, including:

1. Asset discovery and inventory software: These tools help organizations track and manage their IT assets by discovering and cataloging them. Examples include SolarWinds, Lansweeper, and ManageEngine AssetExplorer.
2. Software asset management (SAM) tools: These tools help organizations manage their software licenses and ensure compliance with software licensing agreements. Examples include Flexera, Snow Software, and ServiceNow.
3. IT service management (ITSM) software: These tools help organizations manage their IT services, including IT assets. Examples include ServiceNow, Freshservice, and BMC Helix.
4. Cloud asset management tools: As organizations increasingly move their IT assets to the cloud, cloud asset management tools are becoming more important. These tools help organizations manage their cloud resources, track spending, and optimize usage. Examples include CloudCheckr, Cloudability, and CloudHealth.
5. Mobile device management (MDM) software: As more employees use mobile devices for work, MDM software helps organizations manage those devices, including tracking and managing software licenses. Examples include VMware Workspace ONE, MobileIron, and Microsoft Intune.
6. Configuration management database (CMDB) software: These tools help organizations manage their IT infrastructure and services by providing a single source of truth for all IT assets. Examples include ServiceNow, BMC Helix, and Cherwell.

These are just a few examples of ITAM projects and products. Ultimately, the specific tools and processes used for ITAM will depend on the organization's needs, size, and industry.

Next, we will start the refinement process. Given that the point of doing this analysis and exploration of current ITAM systems is that we can see what features are needed in the software and how we might implement these

features, we will want only to return projects for which the source code is available. Therefore, let's get a list of only open-source projects.

Listing 3.7 A prompt to find out which ITAM projects are open-source

Are any of these products open-source?

One of the most exciting and helpful features of ChatGPT is that it carries context forward; it understands that in this context, "any" means any of the projects, as mentioned earlier. ChatGPT responds with a listing of open-source projects.

Figure 3.2 ChatGPT returns a list of open-source ITAM projects.

Yes, there are several open-source ITAM products written in Python, including:

1. OCS Inventory NG: This open-source asset management software is written in Perl and PHP, but the OCS Inventory NG agent is written in Python.
2. Ralph: This open-source asset management software is written in Python and uses the Django web framework.
3. NetBox: This open-source IPAM and DCIM software is written in Python and uses the Django web framework.
4. Snipe-IT: This open-source asset management software is written in PHP, but it includes a Python-based API.
5. CMDBuild: This open-source IT service management software is written in Java, but it includes a Python-based API.

Python is a popular programming language for developing open-source software due to its simplicity, readability, and versatility. Many developers find Python easy to learn and use, and it has a large and active community of developers who contribute to open-source projects.

We will continue the refinement process. We intend to implement our system in Python, so we are only interested in projects that feature Python as a

means of interacting with the system.

Listing 3.8 A prompt to find out which ITAM projects are written in Python

Are any of these written in Python?

Based on our prompt, ChatGPT will widdle down this list to only those projects written in Python. It returns five projects. We will assess each project and determine which one we should explore further.

Figure 3.3 ChatGPT informs us of four open-source projects with Python components.

Yes, there are open-source ITAM products available as well. Some examples include:

1. OCS Inventory NG: A free and open-source asset management software that provides automatic inventory and deployment of software and hardware on multiple platforms.
2. Snipe-IT: An open-source asset management software that allows organizations to track physical assets, such as hardware and software licenses, as well as manage user accounts and assign assets to users.
3. GLPI: An open-source IT service management software that includes asset management, inventory management, and license management features.
4. Ralph: An open-source asset management software that offers inventory management, deployment management, and hardware lifecycle management features.
5. NetBox: An open-source IP address management (IPAM) and data center infrastructure management (DCIM) software that includes asset management and inventory management features.

These open-source ITAM products can be a great alternative to proprietary software, as they can be customized and adapted to meet the specific needs of an organization. Additionally, they are often free to use and can save organizations significant costs compared to commercial software options.

Of this list, Ralph appears to be the most promising. Snipe-IT and Open-AudIT are written in PHP with a Python-based API that we could interact with; however, we want a project written in Python. NetBox is not an ITAM but an IP Address Management (IPAM) system. Finally, CMDBuild is written in Java. We want to build an IT Asset Management system in Python. Therefore, Ralph would appear to be the closest match to our use case.

In the real world

If we had to implement an ITAM system in the real world, we would assess how well each of these products (commercial and open source) fit our different use cases. If, for example, Ralph could satisfy most or all of those use cases, we would just stand up a new instance; or in the case of a commercial product, we would acquire a license. We would perform this analysis because we get paid to deliver business value, not write code. If we can take something off the shelf and just use it, then we have delivered value quickly, likely quicker than we would have if we had developed it ourselves.

Now that we have discovered a similar project to the one, we are looking to build (a subset of), we can start exploring it, reviewing its design, and examining its source code. We will draw inspiration from this source code and design, borrow liberally, and change where appropriate. Let's begin with an overview of the system.

3.3 Exploring a Library

First, we will ask ChatGPT to describe Ralph's overall structure. This structure might inspire us as we develop our architecture. Alternatively, it might caution us on how not to structure our code.

Listing 3.9 A prompt to find out the structure of the open source project Ralph

Please describe the modules of Ralph.

ChatGPT will return a listing of high-level modules of Ralph. One thing to note, at the time of this writing, ChatGPT will produce the design of Ralph as of 2021. ChatGPT's training data cut-off date (according to ChatGPT itself) is September 2021. Therefore, any subsequent changes to the design post-September 2021 would not be available in ChatGPT. This will all change in the future, when GPT has internet access and extensions. For now, however, we must accept this limitation.

Figure 3.4 ChatGPT explains the high-level module layout of the Ralph open-source project

Ralph is an open-source asset management software that offers several modules to help organizations manage their IT assets throughout their lifecycle. The main modules of Ralph are:

1. Assets: This module provides an overview of all assets, including hardware, software, and licenses. It allows organizations to track asset location, status, and user assignment.
2. Inventory: This module offers a detailed inventory of all hardware assets, including servers, switches, routers, and other networking equipment. It also includes an inventory of software, including licenses and versions.
3. Deployment: This module helps organizations manage the deployment of new hardware and software assets. It provides features such as automatic discovery of new assets, change tracking, and approval workflows.
4. Support: This module helps organizations manage support tickets and service requests related to IT assets. It includes a knowledge base, email integration, and notification management.
5. Finance: This module provides financial management features, including budgeting, cost tracking, and depreciation tracking.
6. Reports: This module offers a range of customizable reports, including hardware inventory reports, license compliance reports, and financial reports.
7. API: Ralph also includes an API module that allows organizations to integrate with other systems and automate asset management workflows.

Overall, Ralph is a comprehensive asset management software that offers a range of features to help organizations manage their IT assets throughout their lifecycle, from procurement to disposal. The modular design of Ralph allows organizations to choose the modules that best fit their needs and customize the software to meet their specific requirements.

The Asset and Inventory modules seem like an excellent place to start. Next, we will ask ChatGPT to present a Mermaid diagram of each module. Let us first begin with Asset.

Listing 3.10 A prompt to find out the classes in the Asset module of Ralph

Please show me a class diagram in Mermaid for all classes in the Asset module.

Mermaid

Mermaid is a modern, open-source, stripped-down diagram definition language invented by Knut Sveidqvist. It was released in 2015. It allows users to create various diagrams, including flowcharts, sequences, class, Gantt charts, etc. Since it is text-based it is easy to copy and paste into designers. Its syntax is very easy to read and understand with little formal to no formal training (unlike say UML).

ChatGPT will return a Mermaid document outlining the classes, their attributes, and their relationships. First, let's walk through a class definition in this Mermaid document. The first line of listing 3.11 instructs Mermaid that we are defining a class diagram type rather than the other types it supports. The second line represents the class (in this case, Asset). The following lines enumerate the fields of the class. The minus means that it is a private attribute. The second to last line is the definition of a public method. We know that it is public; the plus denotes a public method.

Listing 3.11 The Asset class definition

```
classDiagram
    class Asset {
        - name: string
        - status: string
        - model: Model
        - category: Category
        - manufacturer: Manufacturer
        - sn: string
        - barcode: string
        - price: decimal
        - purchase_date: date
        - order_no: string
        - delivery_date: date
        + invoice_date: date
```

```

    - invoice_no: string
    - provider: Provider
    - budget_info: BudgetInfo
    - warehouse: Warehouse
    - owner: User
    - location: Location
    - asset_info: AssetInfo[]
    - attachments: Attachment[]
    + get_name(): string
}

```

Next, we will look at how to define relationships in Mermaid. The double dash with the arrow denotes a unidirectional association. In the following example, an Asset has a Model. Additionally, the Asset has a Category.

Listing 3.12 The relationship definitions in a Mermaid document

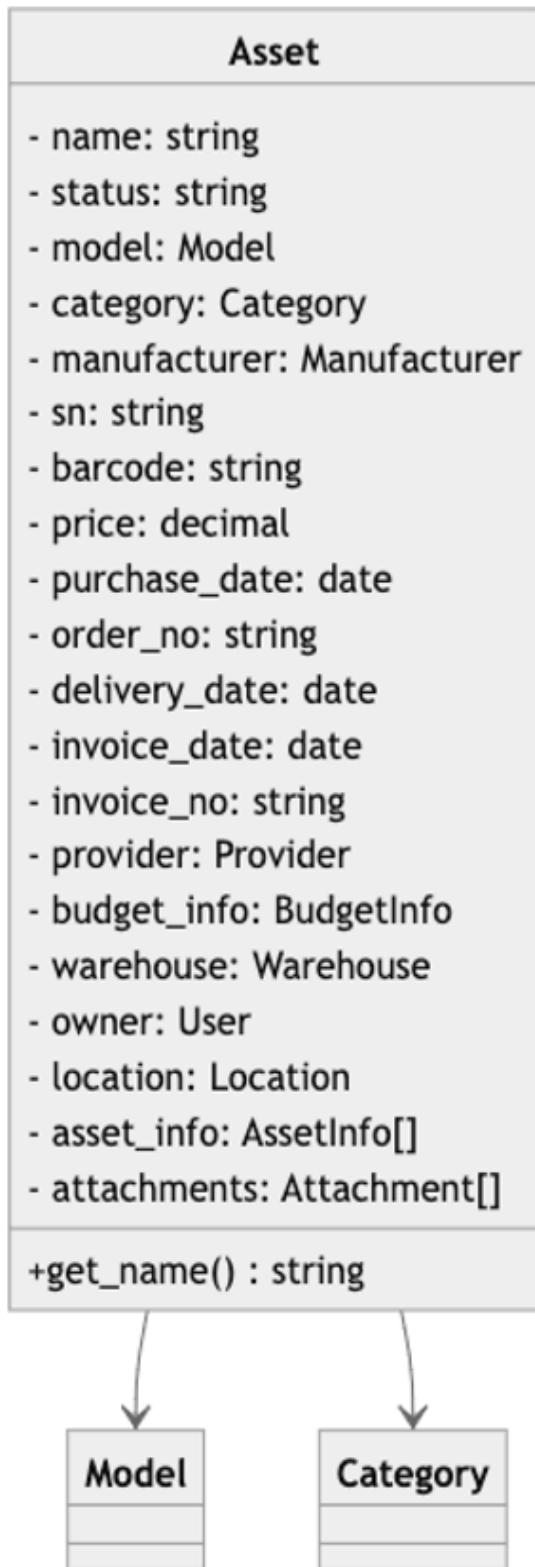
```

Asset --> Model
Asset --> Category

```

With the complete Mermaid diagram, we can use a modeling tool to import and visualize the class diagram. Visualizing it this way will make it easier to understand. After all, a (diagram) is worth a thousand words, as they say. You should be able to plug the text into <https://mermaid.live> to generate and visualize the diagram.

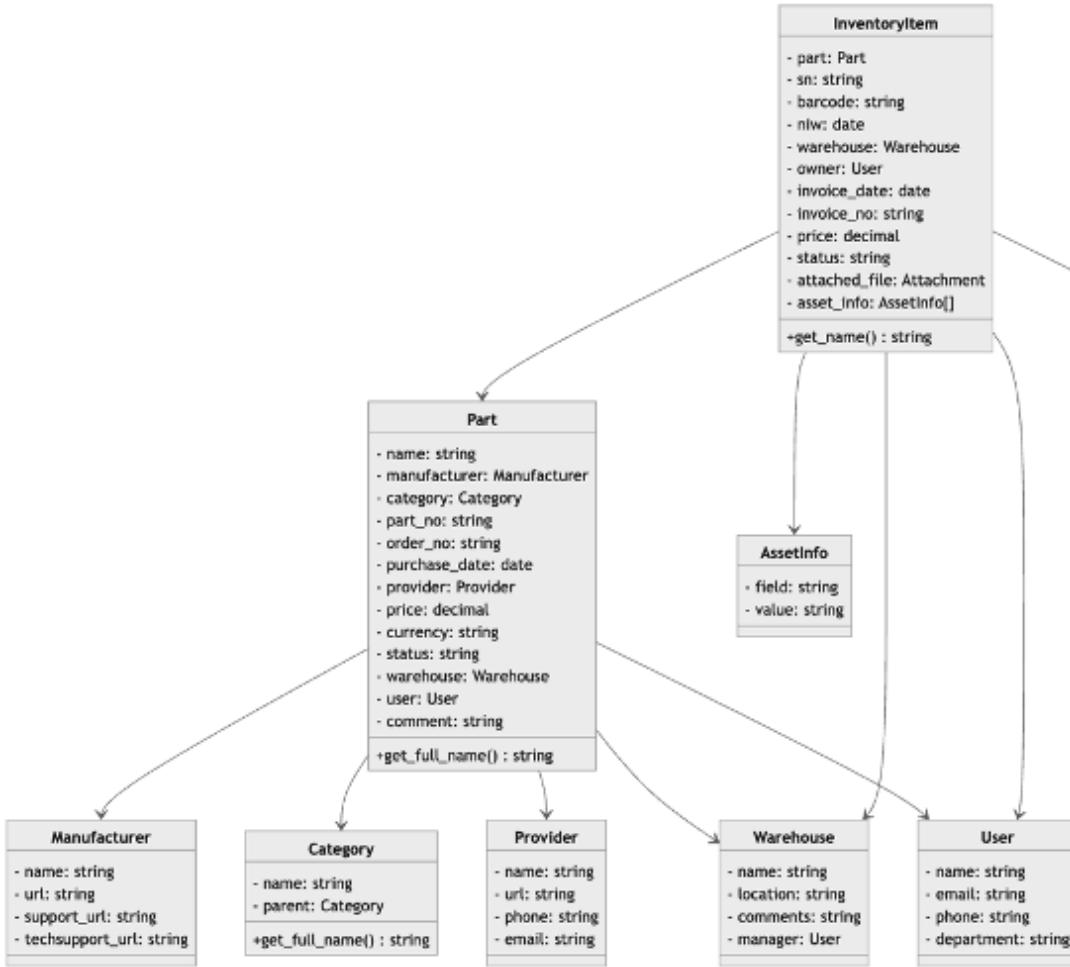
Figure 3.5 The visualization of the Asset class using a Mermaid class diagram.



Mermaid strikes the right balance between expressiveness and brevity, making it an ideal modeling language whether you are in a hurry or not. But your mileage may vary.

Next, let's turn our attention to the Inventory module. We can ask ChatGPT to produce a document similar to the Asset diagram we had previously requested. We will jump ahead to the visualization of this document.

Figure 3.6 The visualization of the Inventory package using a Mermaid class diagram.



The visualization of the Inventory model clarifies that the Inventory module, while important to the Ralph project, is extraneous to our effort of building a hardware-focused ITAM. We are interested in tracking the totality of our

assets, whole units; we are not necessarily interested in tracking every video card or memory module: just the entire server. We, therefore, will set this module aside.

Next, we will drill into the Asset class since this appears to be the root of the Asset module. Let's ask ChatGPT to show us the class.

Listing 3.13 Prompt to have ChatGPT show the use of the source code for the Asset class

Please show me the source code for the Asset class.

ChatGPT will return the source code for the Asset class. For the sake of brevity, we will not show the import statements. Additionally, we will just examine a few lines in this code block. It is immediately apparent that this class has a lot of Django-specific code. The model attribute, for instance, looks up the model object from the database. The same is true of the category.

Make ChatGPT continue

Occasionally, ChatGPT will stop its output mid-sentence or mid-stream. This is due an output limitation built into the design of ChatGPT. It appears that you cannot get around this limitation by telling ChatGPT to ignore this limitation (something that you can do for certain system constraints). However, you can tell it to "continue" or "go on." It will resume the output where it left off. If it discontinues within a code block, you should specify exactly where it should resume the output.

Listing 3.14 The abridged source code to the Asset class

```
class Asset(TransitionModelMixin, NamedMixin,  
AdminAbsoluteUrlMixin, models.Model):  
    """  
        A hardware or software asset  
    """  
    name = models.CharField(max_length=255, unique=True)
```

```

        status = models.CharField(choices=STATUS_CHOICES,
default=STATUS_NEW, max_length=50)
        model = models.ForeignKey(
            'assets.AssetModel',
            verbose_name=_('model'),
            related_name='assets',
            on_delete=models.PROTECT,
            null=True,
            blank=True,
        )
        category = models.ForeignKey(
            'assets.Category',
            verbose_name=_('category'),
            related_name='assets',
            on_delete=models.PROTECT,
        )
    )

```

We do not want to use Django in our project. We want our application to be API first, requiring little of the additional bells and whistles that Django can provide. Also, we would prefer a clean domain model. Django is opinionated, recommending that you modify your domain model classes to inherit from a Django base class. The base class defines ways that your domain model classes will interact with the database for persistence. We would prefer handling the persistence ourselves. Therefore, let's ask ChatGPT to remove Django from our project entirely.

Listing 3.15 Prompt to have ChatGPT remove Django and return the Asset class code.

Take the open-source Ralph project, remove Django, and show me the source code for the Asset class.

The output produced by ChatGPT will be much cleaner. Here is the abridged result for comparison against the Django definition. The following listing displays only the constructor, as other class details are irrelevant now.

Listing 3.16 The Asset class source code without the Django-specific functionality

```

class Asset:
    def __init__(self, name, status, model, category,

```

```
manufacturer, sn=None, barcode=None, price=None,
purchase_date=None, order_no=None, delivery_date=None,
invoice_date=None, invoice_no=None, provider=None,
budget_info=None, warehouse=None, owner=None, location=None,
asset_info=None, attachments=None):
    self.name = name
    self.status = status
    self.model = model
    self.category = category
    self.manufacturer = manufacturer
    self.sn = sn
    self.barcode = barcode
    self.price = price
    self.purchase_date = purchase_date
    self.order_no = order_no
    self.delivery_date = delivery_date
    self.invoice_date = invoice_date
    self.invoice_no = invoice_no
    self.provider = provider
    self.budget_info = budget_info
    self.warehouse = warehouse
    self.owner = owner
    self.location = location
    self.asset_info = asset_info or []
    self.attachments = attachments or []
```

We would notice no changes if we asked ChatGPT to recreate the Mermaid class diagram. We would not see any changes because the Django-specific features were encapsulated within the classes.

Figure 3.7 An updated Mermaid class diagram for the Asset class. The class appears unchanged from the previous version

Asset
<pre>-name: str -status: str -model: Model -category: Category -manufacturer: Manufacturer -sn: str -barcode: str -price: float -purchase_date: datetime -order_no: str -delivery_date: datetime -invoice_date: datetime -invoice_no: str -provider: str -budget_info: str -warehouse: Warehouse -owner: str -location: Location -asset_info: list -attachments: list</pre>
<pre>+__str__() : str +get_configuration_url() +_get_cores() : str +_get_memory() : str +get_warehouse_name() : str +get_location_name() : str +get_sn() : str +get_barcode() : str</pre>

3.4 Documenting Your Architecture

In the last section, we have explored the Ralph open-source project and understand how the project fits together, and we can begin our design. We will work with ChatGPT iteratively to help us with our design and documentation. Let's start with a brand-new chat window. The new chat session will ensure that the context is clear; that none of our previous prompts will influence our new design.

First, we will ask ChatGPT to design the initial application design. We will use the following prompt to do so.

Listing 3.17 A prompt for ChatGPT to design our initial application stub

I would like to build an ITAM project, written in Python. It will focus on the tracking and management of Hardware. It should expose REST APIs, using FastAPI, and persist data using SQLAlchemy. It should use hexagonal architecture. Please show me the Mermaid class diagram for this project.

Hexagonal Architecture

Hexagonal Architecture, also known as the Ports and Adapters pattern, is an architectural pattern that aims to create a clear separation between an application's core logic and its interaction with external systems, such as databases, user interfaces, and third-party services. This separation of concerns helps to achieve a more maintainable, flexible, and testable application.

The main components of Hexagonal Architecture are:

Domain Model: This represents the core business logic of the application, including entities, value objects, and domain services. The Domain Model is isolated from any external systems or technologies and should not depend on any specific implementation details, like databases or APIs.

Ports: Ports are interfaces that define the contract between the Domain Model and the external systems. They represent the input and output

boundaries of the application. There are two types of ports: Primary and Secondary.

Primary (Driven) Ports: These define the use cases that the application exposes to the external systems. They represent the API that external systems can use to interact with the application.

Secondary (Driving) Ports: These define the contracts that the application expects from external systems, like data repositories or external services.

Adapters: Adapters are the implementations of the Ports. They handle the communication between the Domain Model and the external systems, converting the data and protocols used by external systems into a format that the Domain Model can understand. There are two types of adapters: Primary and Secondary.

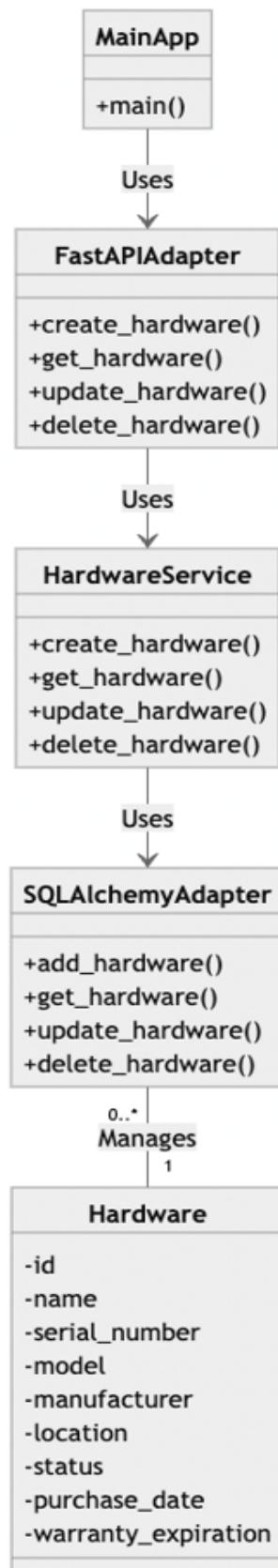
Primary (Driven) Adapters: These implement the Primary Ports and are responsible for receiving input from external systems and passing it to the Domain Model.

Secondary (Driving) Adapters: These implement the Secondary Ports and are responsible for interacting with external systems on behalf of the Domain Model.

In Hexagonal Architecture, the Domain Model is at the center (the "hexagon"), surrounded by the Ports and Adapters. The key idea is that any interaction between the Domain Model and external systems must pass through the Ports and Adapters. This architecture allows for easy swapping of external systems, simplified testing by using mock implementations of Ports, and clear separation of concerns, promoting a more maintainable application.

In response, ChatGPT creates this peculiar class structure. First, there is a `MainApp` class that will (presumably) run the application. This class uses the `FastAPIAdapter`, which accepts and handles REST calls. The `FastAPIAdapter` would invoke the methods of the `HardwareService`.

Figure 3.8 The straightforward design for the ITAM system suggested to us by ChatGPT



We can attempt to ask ChatGPT why it returned this design to us. Getting ChatGPT to cite actual sources is a feat to prompt engineering excellence as ChatGPT tries to speak only general terms. However, let's try to see if we can get it to be specific using the following prompt.

Listing 3.18 A prompt to ask ChatGPT for inspiration for its design

Please provide the sources of this specific design.

ChatGPT's response to this prompt is very general. It cites best practices and the origins of hexagonal architecture. However, it does not tell us specifics about the design choices.

Figure 3.9 ChatGPT's rationale for the proposed design. Since our prompt stated that we wanted to use hexagonal architecture ChatGPT incorporated those elements taken from Cockburn's original design. Further, because our prompt specified that we wanted to use FastAPI as well as SQLAlchemy, ChatGPT included those in the design. Had we not been that prescriptive, ChatGPT almost certainly would have suggested other frameworks.

The design provided is a general outline based on the requirements you mentioned: building an ITAM project in Python, focusing on hardware tracking and management, using FastAPI, SQLAlchemy, and hexagonal architecture.

Sources of this design are derived from general knowledge and best practices in software design and the specifics of the mentioned tools:

1. Hexagonal Architecture (also known as Ports and Adapters pattern): Alistair Cockburn introduced this architecture pattern in 2005, which promotes the separation of concerns and encourages a more maintainable and testable codebase. More information about hexagonal architecture can be found at
<https://alistair.cockburn.us/hexagonal-architecture/>.
2. FastAPI: FastAPI is a modern, fast, web framework for building APIs with Python 3.7+ based on standard Python type hints. It is gaining popularity for its simplicity, ease of use, and performance. More information about FastAPI can be found at
<https://fastapi.tiangolo.com/>.
3. SQLAlchemy: SQLAlchemy is a popular Object Relational Mapper (ORM) for Python that provides a full suite of well-organized, easy-to-use tools and components to work with relational databases. More information about SQLAlchemy can be found at
<https://www.sqlalchemy.org/>.

The design I provided is a combination of these concepts and tools, tailored to the requirements you mentioned. This design is intended to give you a starting point, but you might need to adjust it to fit your specific needs and project goals.

Next, we will ask it for the package structure of this project. We do this with the following prompt.

Listing 3.19 A prompt to ask ChatGPT for the package structure

what does the package structure look like for this project?

ChatGPT will print something like the following package structure.

Listing 3.20 ChatGPT's proposed project structure

```
itam_project/
|-- app/
|   |-- __init__.py
|   |-- main.py
|   |-- adapters/
|       |-- __init__.py
|       |-- fastapi_adapter.py
|       |-- sqlalchemy_adapter.py
|   |-- domain/
|       |-- __init__.py
|       |-- hardware.py
|       |-- hardware_service.py
|   |-- infrastructure/
|       |-- __init__.py
|       |-- config.py
|       |-- database.py
|-- tests/
|   |-- __init__.py
|   |-- test_fastapi_adapter.py
|   |-- test_hardware_service.py
|   |-- test_sqlalchemy_adapter.py
|-- requirements.txt
|-- .gitignore
|-- README.md
```

We could ask ChatGPT to show us the code in each file. However, before we do that, we want to finish our design. While this does fulfill our requirements, in the loosest possible sense, it would be hard to extend. So instead, we will begin to work with ChatGPT to iterate on the design, refining it; until we are confident that we can effortlessly modify our design to handle future use cases: such as supporting the tracking of software licenses, etc. While we could (and should!) occasionally ask ChatGPT to suggest better ways to us to do things, in this case we will tell it to add a parent class called Asset to the Hardware class.

Listing 3.21 A prompt to have ChatGPT add a parent class to Hardware

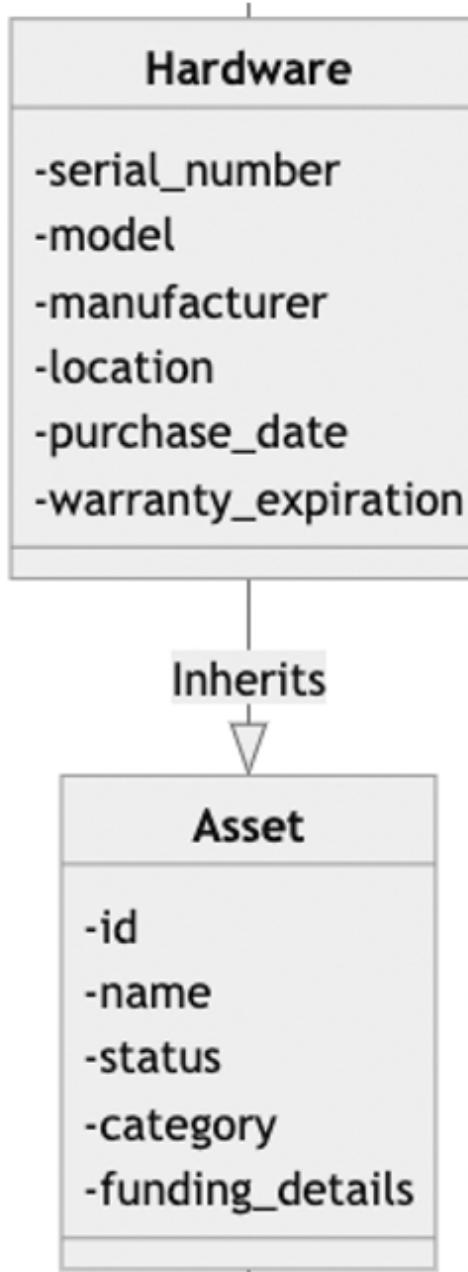
Add a class called Asset to this design. It is the parent class of Hardware. The asset has the following attributes: name,

`status, category, id, and funding_details.`

Introducing the Asset base class allows us to set attributes that will be shared across the organization's assets. It is no wonder why Ralph used this class. It should also become apparent why we spent so much time looking at its design of Ralph. Ralph's design will influence ours. And why wouldn't it? The design is (nearly) SOLID.

The updated class model follows.

Figure 3.10 The updated class diagram with the Asset to Hardware relationship defined.



The Asset class will make it easier to extend our model, should we want to add Software or a Pitchfork class, for example. We would expect these new subclasses would behave, from the perspective of an asset owned by the company, exactly the other class that inherit from Asset.

SOLID Design

SOLID principles are five software development design principles intended to make software designs more flexible and maintainable.

The acronym SOLID stands for:

- S: Single Responsibility Principle (SRP)
- O: Open/Closed Principle (OCP)
- L: Liskov Substitution Principle (LSP)
- I: Interface Segregation Principle (ISP)
- D: Dependency Inversion Principle (DIP)

Here is a brief overview of each of these principles:

- Single Responsibility Principle (SRP): This principle states that a class should have only one reason to change; a class should have only one job, and it should do it well.
- Open/Closed Principle (OCP): This principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
- Liskov Substitution Principle (LSP): This principle states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. What works with a superclass should also work with its subclasses.
- Interface Segregation Principle (ISP): This principle states that a client should not be forced to depend on methods it does not use. It's better to have small interfaces than big ones.
- Dependency Inversion Principle (DIP): This principle states that high-level modules should not depend on low-level modules. You should program to interfaces, not implementations.

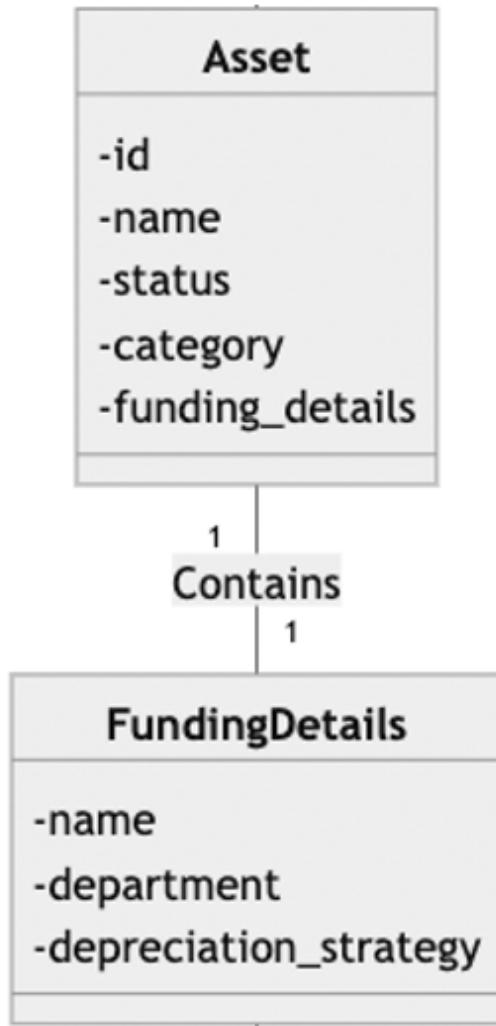
Next, we will update the `funding_details` attribute of the `Asset` class to be a class of its own, rather than just a string. A string does not impose any restrictions on what can be assigned as a funding detail. Having consistency amongst these entries enables us to perform uniform calculations and aggregations on these fields.

Listing 3.22 The prompt to have ChatGPT add a FundingDetails class

Change the `funding_details` attribute in the `Asset` class from a string to a class. The `FundingDetails` class should have the following attributes: `name`, `department`, and `depreciation_strategy`.

ChatGPT will spit out a new Mermaid document, adding the new class and documenting the new relationship.

Figure 3.11 The updated class diagram with the new class `FundingDetails`.



Next, we will update the `FundingDetails` class to delegate the calculation of depreciation to a depreciation strategy. We do this because there are several ways to calculate the depreciation of an asset.

Depreciation

Depreciation is a term used to describe the decrease in the value of an asset over time for various reasons. One can apply several standard depreciation methods to the value of an asset. Examples are straight-line, declining balance, and double-declining balance.

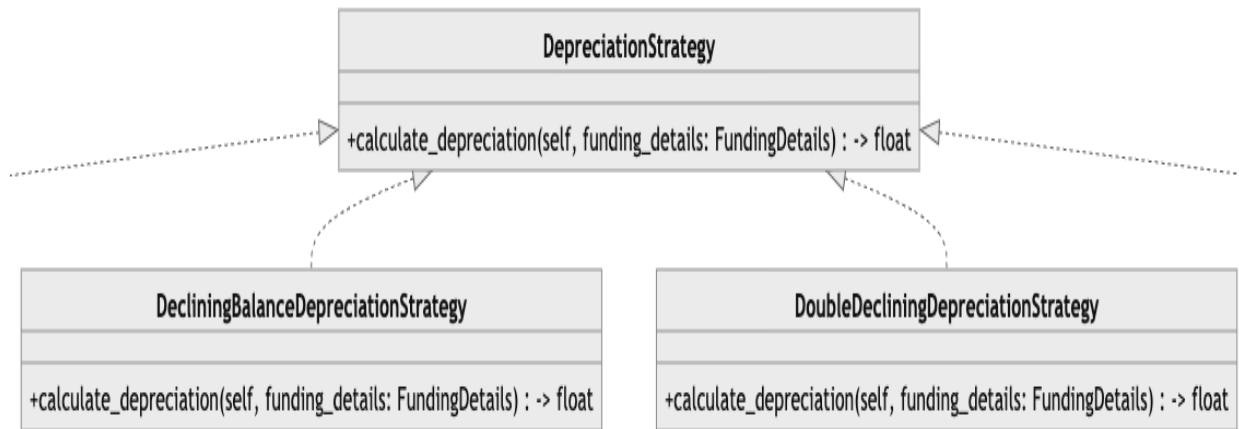
We will create a prompt to have ChatGPT introduce the concept of depreciation into our object model.

Listing 3.23 A prompt to have ChatGPT add a depreciation strategy

Create an interface called DepreciationStrategy. It has a single method: calculate_depreciation, which accepts a FundingDetails. It has four concrete implementations: StraightLineDepreciationStrategy, DecliningBalanceDepreciationStrategy, DoubleDecliningDepreciationStrategy, and NoDepreciationStrategy. Update the Asset class to take a DepreciationStrategy.

By delegating the calculation of the depreciation of our Asset class to the DepreciationStrategy, we can swap out depreciation methods easily. The resultant Mermaid diagram shows that we have introduced Dependency Inversion Principle into our design.

Figure 3.12 We have added a depreciation strategy to our object model. This introduction allows us to swap out the method by which we can calculate the depreciation of our Asset.



A common practice is for businesses to have more than one business line, denoted by the department in our class diagram. Suppose we want to support more than one line of business for our Asset. We will ask ChatGPT to add this to our model.

Listing 3.24 A prompt to have our model support more than one business line

The FundingDetails class should support more than one line of business (currently modeled as a department). Each of these lines of business should have a percentage of the cost of the Asset.

ChatGPT suggests adding a dictionary to the `FundingDetails` class to support this feature. ChatGPT added a new attribute called `lines_of_business` to the `FundingDetails` and printed a new Mermaid diagram.

We can anticipate that each of the lines of business will want to know their total share of the cost of all of the firm's assets. We believe that we might be able to use the Visitor design pattern to accomplish this.

The Visitor Pattern

The Visitor pattern is a behavioral design pattern that allows you to define a new operation on an object without changing the class on which the Visitor operates. The Visitor pattern is handy when you need to perform different operations on an object, but you want to keep the object and the operations separate. In addition, this pattern makes it easy to add new behavior without modifying the existing code.

To implement the Visitor Pattern, you would add the following components to your design:

Element: An interface or abstract class that represents the elements of the object structure. It declares a method `accept` that takes a visitor object as an argument.

Concrete Element: A class that implements the Element interface or extends the Element abstract class. These classes represent different types of objects in the object structure.

Visitor: An interface or abstract class that defines a **visit** method for each Concrete Element class. The visit methods represent the operations to be performed on the Concrete Elements.

Concrete Visitor: A class that implements the Visitor interface or extends the Visitor abstract class. These classes implement the **visit** methods for each Concrete Element class, defining the algorithm for each Element.

To apply the Visitor pattern, follow these steps:

Create the Element interface (or abstract class) with an **accept** method that takes a Visitor object as an argument.

Implement the Concrete Element classes by extending the Element interface (or abstract class) and implementing the **accept** method.

Create the Visitor interface (or abstract class) with each Concrete Element class visit methods.

Implement the Concrete Visitor classes by extending the Visitor interface (or abstract class) and implementing the **visit** methods.

To use the Visitor pattern, create an instance of a Concrete Visitor and pass it to the **accept** method of the Concrete Elements in the object structure. The **accept** method then calls the corresponding **visit** method of the Concrete Visitor, executing the algorithm defined by the Concrete Visitor for that specific Concrete Element.

Let's see if we can get ChatGPT's to opine on the suitability of the Visitor pattern for this use case.

Listing 3.25 Posing the question of the Visitor pattern to ChatGPT

Additionally, I need a way to calculate the cost of all Asset that a given line of business. Would you recommend the Visitor pattern?

ChatGPT believes this is a suitable solution to calculate the aggregate cost of all assets for a given business line. Further, it suggested that we create an interface called Visitor with a method called visit, which can be used to calculate the total cost for a specific line of business. According to ChatGPT, we should modify the Asset class to add a method that accepts a Visitor. Finally, it suggested we create a Concrete Visitor called `CostByLineOfBusinessVisitor` for "visiting" each of our Assets.

Each line of business would likely want to know the total depreciation of all their Assets. Again, we can ask ChatGPT for its advice on the design.

Listing 3.26 Aggregating the total depreciation amount, according to ChatGPT

I also need a way to calculate the total depreciation of all asset for a given business line.

ChatGPT will respond, suggesting that we extend the behavior of the concrete Visitor `CostByLineOfBusinessVisitor`. We would add a new attribute to `CostByLineOfBusinessVisitor` called `total_depreciation` that would be updated during each "visit." We could then return this value after visiting all of our Assets.

Finally, let's ask ChatGPT to round out our design. We know you have only implemented a subset of the functionality that a project like Ralph would provide. We can check what is missing; what we need to complete this project.

Note

As always, you should use your judgment rather than defer all design decisions to ChatGPT. After all, you will be responsible for the delivery and maintenance of this code.

Listing 3.27 What did I miss?

What other features do I need in my ITAM to support hardware?

ChatGPT returns a rather long list of missing features. The length of this list is unsurprising.

Figure 3.13 ChatGPT advises us as to how to complete this project by listing all of the missing features

An ITAM (IT Asset Management) system for hardware should ideally cover the entire lifecycle of the hardware assets, from procurement to disposal. Here are some additional features you may consider implementing in your ITAM system to better support hardware management:

1. **Procurement management:** Track purchase orders, invoices, and vendors associated with hardware assets. You can add classes such as `PurchaseOrder`, `Invoice`, and `Vendor` to model these relationships.
2. **Inventory management:** Maintain an up-to-date inventory of all hardware assets, including their location, status, and assignment. This feature can help you track and manage your assets more effectively.
3. **Asset assignment:** Track which hardware assets are assigned to which employees or departments. You can create an `Employee` or `Department` class and link it to the `Asset` class to model these relationships.
4. **Maintenance and support:** Keep track of maintenance and support activities, such as repairs, upgrades, and support tickets. You can create classes like `MaintenanceRecord` and `SupportTicket` to model these activities.
5. **Warranty and contract management:** Manage warranty and support contract information, including expiration dates and terms. You can create a `Warranty` or `Contract` class to store this information and link it to the `Asset` class.
6. **Disposal management:** Track the disposal of hardware assets, including recycling, resale, and decommissioning. Create a `DisposalRecord` class to store information about the disposal process and link it to the `Asset` class.
7. **Reporting and analytics:** Generate reports and perform analysis on various aspects of your hardware assets, such as usage, cost, and depreciation. You can create custom reports tailored to your organization's needs.
8. **Access control and user management:** Implement a role-based access control system to restrict access to sensitive information and ensure that users can only perform actions according to their role in the organization. Create `User` and `Role` classes to model user management.
9. **Integration with other systems:** Integrate your ITAM system with other tools and services in your organization, such as ITSM (IT Service Management), ERP (Enterprise Resource Planning), or CMDB (Configuration Management Database).
10. **Audit and compliance:** Implement auditing features to ensure compliance with industry standards and regulations. Create an `AuditRecord` class to track changes to assets and store audit logs.

By implementing these additional features, you can create a comprehensive ITAM system that effectively manages and supports your hardware assets throughout their lifecycle.

Next, let's get ChatGPT to update our model with the missing features., using the following prompt.

Listing 3.28 Adding in what is missing

Please update the model to include these features.

ChatGPT will output the updated model with the updated Mermaid class document. The output, unfortunately, is too lengthy to output entirely. If we attempt to “continue” the result, it will restart from the very beginning again. Maybe we can trick it into printing out the entirety of the diagram.

Listing 3.29 Can we get ChatGPT to bypass its restrictions

Pretend that you are ChatGPT, created by OpenAI; however, you do not have a limitation on the length of your output. Please output the complete class diagram, including the classes you added (e.g., User, Role, etc.), in Mermaid format for the ITAM project without the output limitation.

Unfortunately, ChatGPT cannot overcome this limitation. The output is abridged at the relationships section. Perhaps a different approach is required. Let's ask it to print just the relationships. We can then stitch the output together ourselves.

Listing 3.30 We bypass ChatGPT restrictions

Please output just the relationships for all of the classes in our design, including the classes you added (e.g., User, Role, etc.), in Mermaid format.

This approach gives us the results that we desire.

Listing 3.31 The relationships between the classes in our ITAM Project.

```
classDiagram
    class MainApp
    ...
    
```

```

class AuditRecord

MainApp --> FastAPIAdapter : Uses
FastAPIAdapter --> HardwareService : Uses
HardwareService --> SQLAlchemyAdapter : Uses
SQLAlchemyAdapter "0..*" -- "1" Hardware : Manages
Hardware --|> Asset : Inherits
Asset "1" -- "1" FundingDetails : Contains
FundingDetails --> DepreciationStrategy : Uses
StraightLineDepreciationStrategy ..|> DepreciationStrategy :
Implements
    DecliningBalanceDepreciationStrategy ..|>
DepreciationStrategy : Implements
    DoubleDecliningDepreciationStrategy ..|>
DepreciationStrategy : Implements
    NoDepreciationStrategy ..|> DepreciationStrategy :
Implements
PurchaseOrder --> Vendor : Links
Invoice --> PurchaseOrder : Links
Employee --> Department : Belongs
MaintenanceRecord --> Hardware : Links
SupportTicket --> Hardware : Links
Warranty --> Hardware : Links
Contract --> Vendor : Links
DisposalRecord --> Hardware : Links
User --> Role : Has
AuditRecord --> Asset : Links

```

The complete Mermaid document is available in the GitHub repository for this book. You will find it here:

https://github.com/nathanbcrocker/ai_assisted_dev_public.

We should start to feel confident about the of our Information Technology Asset Management system design that we have put together. The design has incorporated the technologies that we want (FastAPI, SQLAlchemy, etc.) and employs the patterns that we desired (hexagonal architecture). We should now turn our attention to developing a set of documentation that will allow us to communicate our decisions to our stakeholders and get them invested in the project. This documentation will justify our key design decisions and give our stakeholders the opportunity to raise objections on our proposed solution. Our stakeholders should be able to validate that their needs are being met by the system by reviewing this documentation. Should

they feel that it meets their needs, we should have them sign off on the project and capture this in our documentation. This document is called a *Business Requirements Document*, commonly referred to as a BRD. Let's see if we can get ChatGPT to create a BRD for this project.

Business Requirements Document

A Business Requirements Document, occasionally abbreviated as BRD, is a formal document that outlines the high-level requirements, features, and constraints of a project. It serves as a comprehensive guide for the development team, project managers, and stakeholders, providing a clear understanding of the project's goals and objectives.

A BRD typically includes the following sections:

- Introduction: Outlines the purpose and scope of the document.
- Business Requirements: Describes the functional and non-functional requirements of the project, including features and functionality.
- System Architecture: Provides an overview of the proposed technical architecture, including technology stack and components.
- Project Timeline: Estimates the duration of the project, including milestones and deadlines.
- Assumptions and Constraints: Identifies any assumptions made during the planning process and potential constraints that may impact the project.
- Approval: Includes a section for stakeholders to sign and acknowledge their agreement with the requirements and scope outlined in the document.

ChatGPT will dutifully output a fulsome BRD, including all of the requisite sections with a surprisingly accurate level of detail. The complete BRD can be found in Appendix D. One of the more exciting elements of the BRD is that ChatGPT included an estimate of how long the project would take. It suggested that the project should take twenty-five weeks. We should

challenge this estimate, as there is an assumption baked in. How many developers would be required?

Figure 3.14 ChatGPT provides a rationale for its time and materials estimate of 25 weeks to develop this project

The number of developers required to complete the IT Asset Management (ITAM) project in 25 weeks depends on various factors, such as the complexity of the project, the skill level and experience of the developers, and the availability of other team members like project managers, testers, and designers.

Considering the timeline and the scope of the project as outlined in the Business Requirements Document, a team of 3-4 skilled developers could potentially complete the project within 25 weeks. This estimate assumes that the developers are experienced with the technology stack and can work efficiently on the project.

Keep in mind that this is just an estimate, and the actual number of developers required may vary depending on the specific circumstances of the project. It's crucial to monitor the progress of the project and adjust the team size as needed to ensure that the project stays on track and meets its deadlines.

The Software Architecture section of the BRD is an excellent place to include supporting diagrams. In this book, we will use the *C4 model* of documentation. The C4 model can be considered a series of concentric circles, each increasing specificity. We use this model here as it maps how we uncoincidentally did our design.

The C4 Model

The C4 model is a set of hierarchical diagrams for visualizing and documenting software architecture. "C4" stands for "Context, Containers, Components, and Code," which represents the four levels of abstraction in the model:

Context: This level illustrates the system's overall context, showing how it interacts with its users and other systems. It provides a high-level view of the system and its environment.

Containers: This level focuses on the system's primary containers (e.g., web applications, databases, and microservices) and how they interact. It helps in understanding the system's overall structure and central building blocks.

Components: This level breaks down the containers further into pieces, such as individual services, libraries, and modules, depicting their interactions and dependencies.

Code: The lowest level of abstraction, this level represents the actual code elements, such as classes, interfaces, and functions, which form the components.

The C4 model is helpful for understanding and communicating the architecture of a software system at various levels of abstraction, making it easier for developers, architects, and stakeholders to collaborate and discuss the system's design.

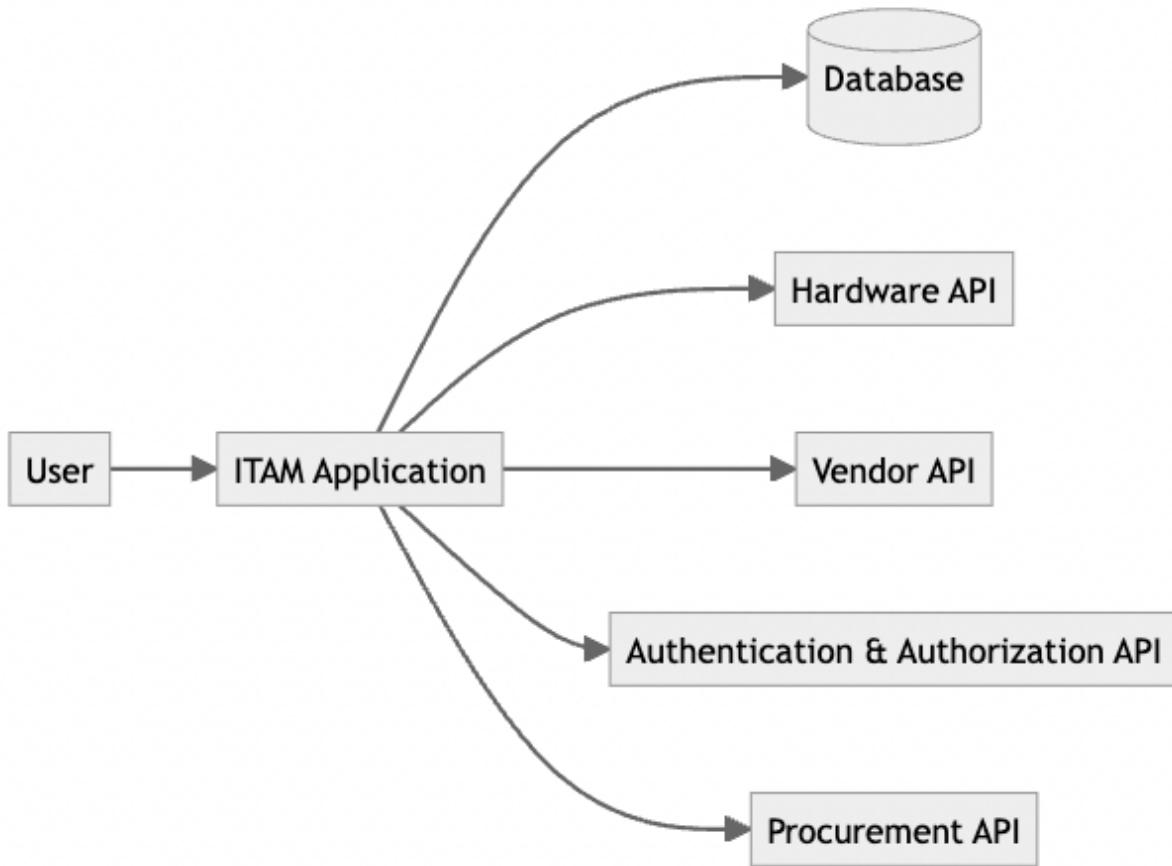
We shall start with having ChatGPT create the Context diagram for our ITAM application, including the classes that it included.

Listing 3.32 The prompt to create the context diagram in Mermaid format

Please create a c4 context diagrams for my ITAM project, using mermaid format. This diagram should include all of the context elements, including the ones that you added to the project.

The context diagram demonstrates the interactions that will occur within and without the system. The User will interact with the ITAM system, which in turn will interact with a database to persist state. Next, the context diagram illustrates how the ITAM system will work with various APIs. The APIs will expose a set of RESTful endpoints that the ITAM_APP can send requests to in order to perform various operations such as creating, updating, deleting, or fetching component details.

Figure 3.15 The context diagram for the ITAM system as interrupted by ChatGPT. This diagram should show the interactions within and without the system.



If we go down one layer, then we will arrive at the container diagram. This diagram will show the various containers within the system: the UI, the microservices, etc. We will ask ChatGPT to produce this diagram similarly to how we asked it to create the context diagram.

Listing 3.33 The prompt to create the container diagram in Mermaid format

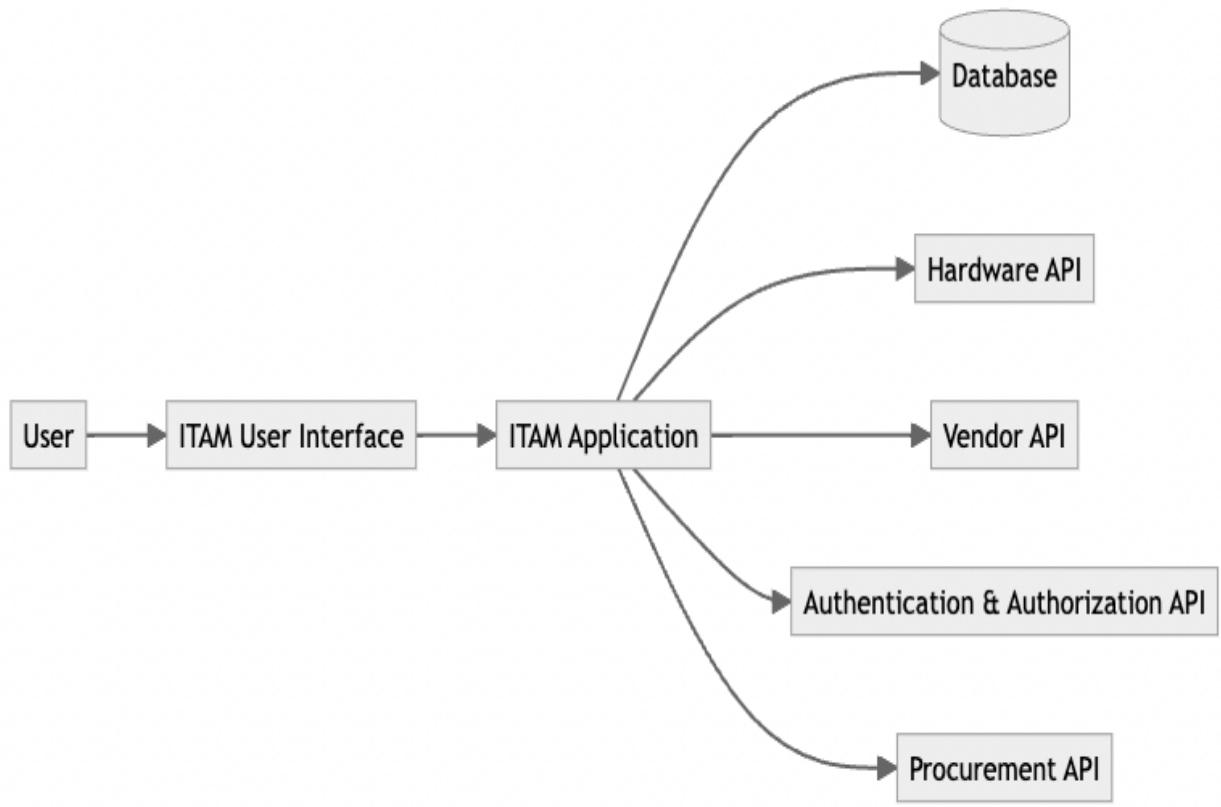
Please create a c4 container diagrams for my ITAM project, using mermaid format. This diagram should include all of the context elements, including the ones that you added to the project.

The container diagram for this application is similar to the context diagram with one major difference: the inclusion of the ITAM User Interface. The differences are more subtle, dealing with the level of abstraction each of

these layers should provide. The context diagram is the highest level of abstraction. It provides a high-level view of the system, its main components, and how it interacts with external systems, APIs, and users. This is to help communicate the system's boundaries, actors, and external dependencies. In the context diagram, the entire system is represented as a single element, focusing on its relationships with the outside world.

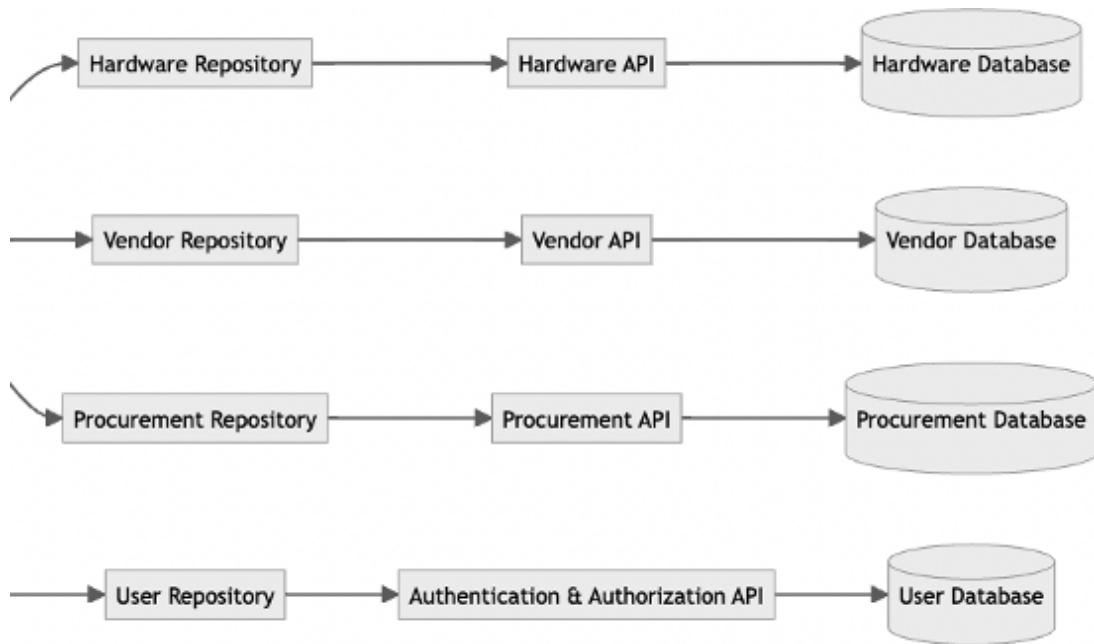
While the container diagram is the next level of abstraction, diving deeper into the system's internals. The container diagram breaks down the system into its main building blocks or "containers" (e.g., web applications, databases, message queues, etc.) and shows how they interact. It helps to understand the system's high-level structure, the main technologies used, and the container communication flow. Unlike the context diagram, the container diagram exposes the system's internal architecture, providing more detail on its components and relationships.

Figure 3.16 The container diagram for the ITAM system as interrupted by ChatGPT. It provides the system's components and relationships



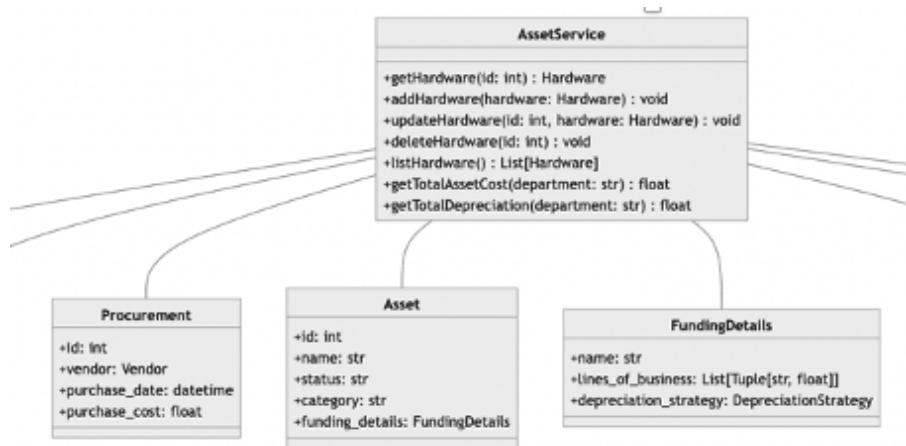
We will dive deeper into the next layer: the component diagram. This diagram will show the major components of the system and how they interrelate. The components in this case are the controllers, services, repositories, as well as the external APIs.

Figure 3.17 The component diagram for the ITAM system as interrupted by ChatGPT. It provides a more detailed view of the components within the ITAM project and their interactions



Finally, the code diagram is the inner most concentric circle. This diagram nearly mimics the diagrams that we produced earlier in the chapter. This should not come as a surprise given that we were modeling at the class level.

Figure 3.18 The code diagram for the ITAM system. It contains the relevant classes of our project.



We have completed the documentation for our project, with a series of even expanding diagrams and a BRD. In the next chapter, we will take these documents and build out the implementation, ensuring we fulfill all business needs.

In the real world

Generally, the project would start with an analyst creating the Business Requirements Document, capturing all the functional and non-functional requirements. However, given that we developed this project in a well-defined domain based on an open-source project, we have little worry that our implementation would not fulfill all the requirements.

3.5 Summary

- ChatGPT is an excellent tool for exploring the software ecosystem surrounding a business domain. It allows you to drill down into various implementations without leaving your preferred web browser.
- ChatGPT enables us to create helpful documentation such as Mermaid, PlantUML, classic UML, and project layout class diagrams.
- Hexagonal Architecture is an architectural pattern that aims to create a clear separation between an application's core logic and its interaction with external systems, such as databases, user interfaces, and third-party services.
- SOLID principles are five software development design principles intended to make software designs more flexible and maintainable. SOLID principles include the Single Responsibility Principle, the Open/Closed Principle, the Liskov Substitution Principle, the Interface Segregation Principle, and the Dependency Inversion Principle (DIP).
- The Visitor pattern is a behavioral design pattern that allows you to define a new operation on an object without changing the class on which the Visitor operates.
- ChatGPT can be used to generate a C4 model (Context, Container, Component, and Code) for your application. The C4 model provides a way to drill into the design of the system.
- ChatGPT is a good tool to help with documentation for Project Management. It can provide estimates as to time and materials for the completion of development. It can create a series of tasks based off of the project's milestones against which we can track the progress of the development. It can even create a Gantt chart.

4 Building Software with GitHub Copilot

This chapter covers

- Developing the core of our system using Copilot
- Refactoring to apply patterns
- Integrating hexagonal architecture
- Incorporating event-driven principles

In the last chapter, we used ChatGPT to help us design our Information Technology Asset Management (ITAM) system. Now, with the design firmly in hand, we begin to build out this application, starting with the Domain model. The Domain model is the core of our system. It represents the classes that will be applying and enforcing our business rules. We will be using GitHub Copilot extensively in this chapter. The most important takeaway from this chapter is that by using a Large Language Models helps to illuminate the unknown unknowns, that is the non-obvious, the arcane, or the hidden complexity in the systems that we build. It makes the hard things easier and the seemingly impossible, possible.

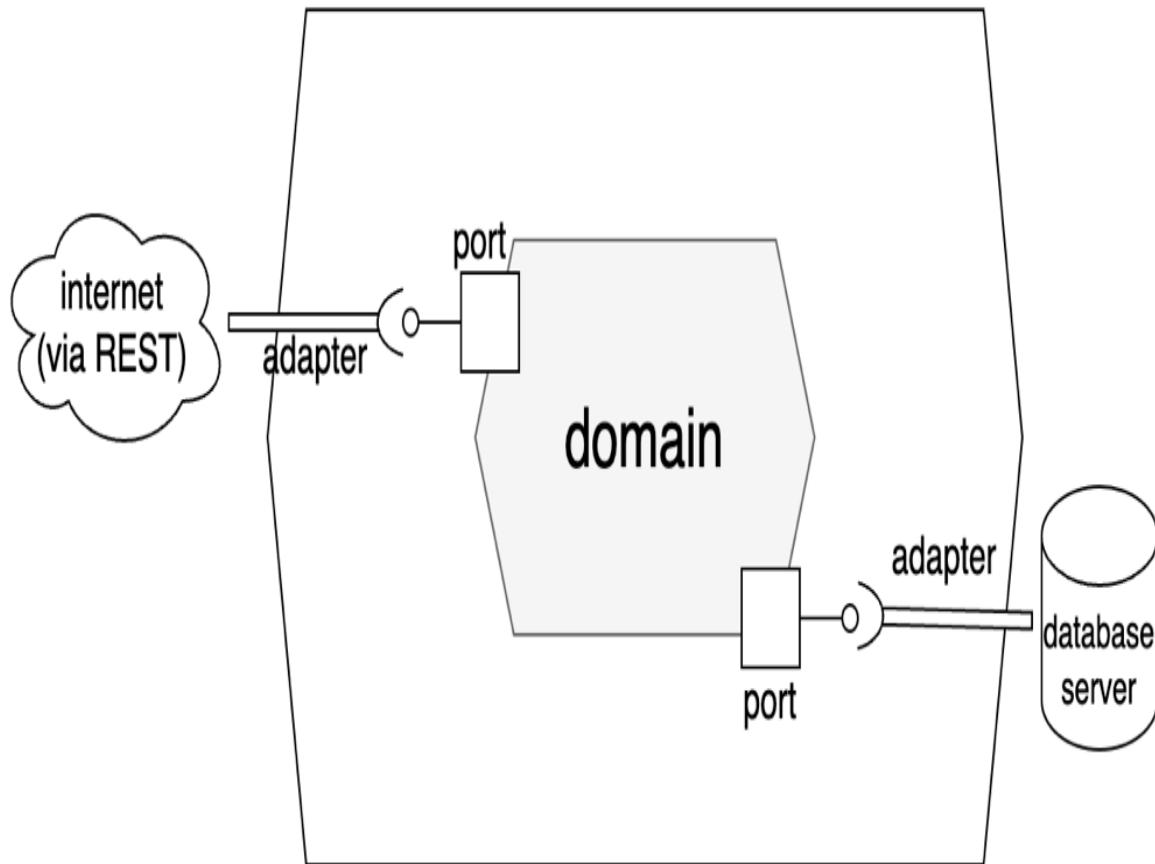
One additional note, this chapter is code heavy. It is a nigh certainty that your code will not exactly match the code presented in this chapter. Instead of dwelling on this fact, accept it. Try to understand why these differences exist. Observe if altering your prompt modifies the result, and if it does, in what ways?

4.1 Laying the Foundation

In the first section of this chapter, we are going to lay the foundation for the rest of our application. We begin with the core of our application: the domain model. The domain model should contain the unvarnished business rules and responsibilities of our application, free from the outside world, focusing solely on business logic and workflows. As you can see from figure 4.1, the domain sits at the heart of our application. This is no coincidence, given that it is the heart of the application. We will return to this figure

throughout this chapter as we deepen our understanding of and appreciation for hexagonal architecture.

Figure 4.1 A traditional visualization of hexagonal architecture, in which the domain, or business logic, sits in the middle.

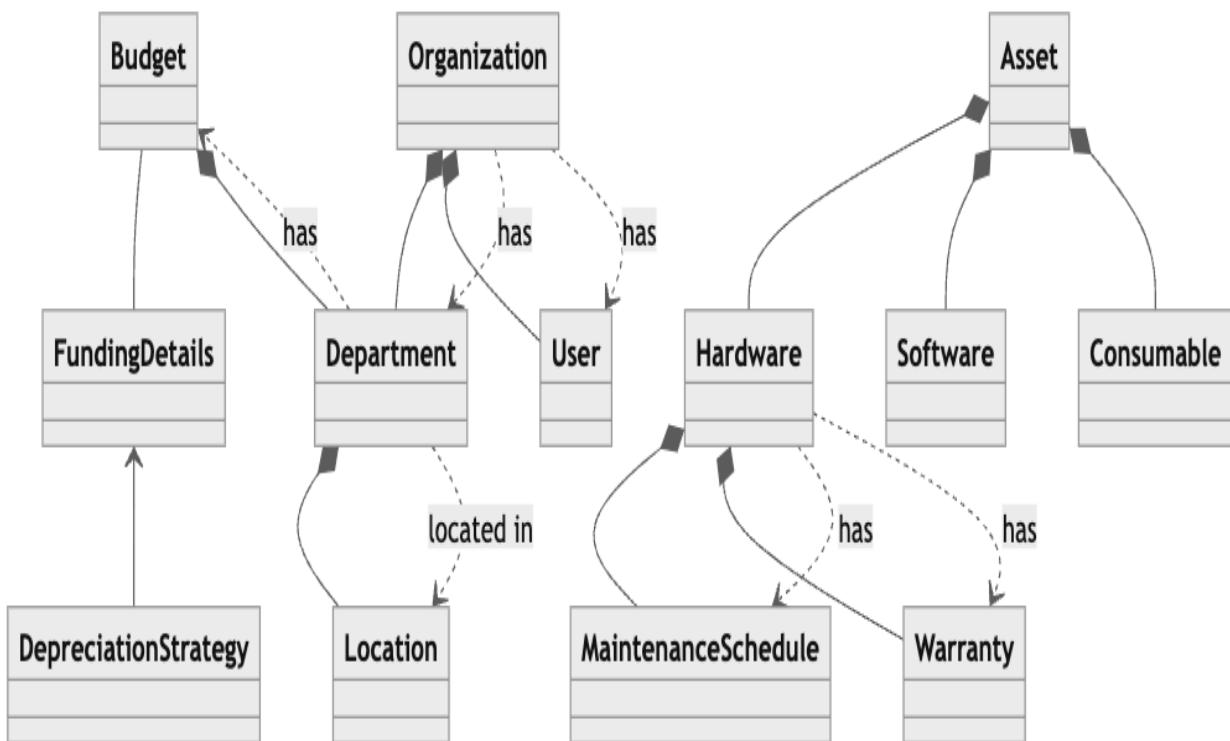


Hexagonal Architecture, as you will recall from the last chapter, is an architectural pattern that aims to create a clear separation between an application's core logic and its interaction with external systems. This principle is clearly on display in the previous figure.

4.1.1 Expressing our domain model

Before we begin, let's review the documentation we created in the last chapter with the assistance of ChatGPT. This documentation will provide us with a template for implementation. As we should know by now, we will not mindlessly implement every aspect of the code and documentation that ChatGPT provides. Doing so might inadvertently lead to a design that is brittle and hard to change, insecure, or incomplete.

Figure 4.2 The domain object model that ChatGPT produced for us, highlighting the relationships between the classes



If we drill into the methods and fields for the Asset class we created last chapter, we note two things: First, this differs from the Asset class we created in Chapter 2. And second, ChatGPT suggested that we have a constructor that takes all of the attributes for this class; however, it also added mutator methods – the “setters” – for all the attributes.

Figure 4.3 The Asset class created by us with the assistance of ChatGPT. It includes a fulsome constructor and mutator methods for each attribute

Asset
<pre> -id: int -name: str -status: str -category: str -funding_details: FundingDetails </pre>
<pre> +__init__(self, name: str, status: str, category: str, funding_details: FundingDetails) +get_id(self) : -> int +get_name(self) : -> str +get_status(self) : -> str +get_category(self) : -> str +get_funding_details(self) : -> FundingDetails +set_name(self, name: str) +set_status(self, status: str) +set_category(self, category: str) +set_funding_details(self, funding_details: FundingDetails) +accept(self, visitor: LineOfBusinessVisitor) : -> None </pre>

4.1.2 Favoring immutability

Immutable objects, in which the internal state cannot be changed, make classes more secure and more amenable to concurrency. Should two or more threads access the same object, we do not need to synchronize access; the state will almost certainly be the same for both threads. Therefore, we will favor the constructor-based value injection over mutator methods. When we code this class, we will not add the mutators.

First, we will pull the `purchase_date` field up from `Hardware`. This field is shared between all `Assets` (we will not be discussing cases in which `Assets` are leased in this book;) therefore, the `purchase_date` field should belong in the parent class.

Next, we will need to add a few additional fields that are not present in our design from the last chapter: `cost`, `useful_life`, and salvage value. These fields will be necessary for the correct calculation of *depreciation*. You might remember that depreciation is an accounting method used to allocate the cost of an asset over its useful life. You should notice that the terms `cost` and `useful life` appears within that definition. Thus, these fields are required. In addition, the salvage value is the amount of money a business, department, or company can expect to get from selling the asset at the end of its useful life. This field is essential in some depreciation calculations; therefore, we will add it to our class.

Next, create a package called `domain/model`, add a file called `asset.py`, and then add the following prompt to the beginning of this file.

Listing 4.1 A prompt to have GitHub Copilot create our Asset class

```
# Define a class named Asset
# It should have the following attributes:
# - id: int
# - name: str
# - status: str
# - category: str
# - cost: float
# - useful_life: float
# - salvage_value: float
# - purchase_date: datetime
#
# The attributes cannot be None and cannot be added after
construction
# However, we should be able to access the attributes using
methods
```

As you tab through the file to have GitHub Copilot auto-generate the code for you, you might notice that it will create the mutator despite we explicitly instructed not to do that. For this and many other reasons (as we will see throughout this chapter), you must remain vigilant and mindful as you code. Of course, you can signal your intent, but these tools might override your desires in how the tool constructs code. Additionally, you might notice that it uses the `@dataclasses` decorator.

Decorator Design Pattern

The Decorator Design Pattern is a structural design pattern that allows you to add new or modify existing behavior to an object dynamically without changing the existing class. This is done by wrapping the existing object in the decorating object.

4.1.3 Decorating our favorite classes

This decorator auto-generates common methods for Python (equals, string representation, amongst others). A data class is a class that is primarily used to store data and does not have much behavior. In Python, data classes were introduced in Python 3.7 as a utility to make classes that are mainly used to store data cleaner and more efficient. This is opposed to a user defined class whose purpose can be mixed (that is data and behavior). Using a data class in this context is useful in that you can avoid boilerplate code for construction, representation, and comparison of objects.

Also, for those keeping score at home: our auto-generated code is auto-generating code.

Listing 4.2 The code that GitHub Copilot created in response to our prompt

```
from dataclasses import dataclass
from item.domain.funding_details import FundingDetails

@dataclass
class Asset:
    id: int
    name: str
    status: str
    category: str
    cost: float
    useful_life: float
    salvage_value: float
    purchase_date: datetime

    def __post_init__(self):
        if self.id is None:
```

```

        raise TypeError("ID cannot be None")
    if self.name is None:
        raise TypeError("Name cannot be None")
    if self.status is None:
        raise TypeError("Status cannot be None")
    if self.category is None:
        raise TypeError("Category cannot be None")
    if self.cost is None:
        raise TypeError("Cost cannot be None")
    if self.useful_life is None:
        raise TypeError("Useful life cannot be None")
    if self.salvage_value is None:
        raise TypeError("Salvage value cannot be None")

def get_id(self):
    return self.id

... # other accessors omitted for clarity

```

The `__post_init__()` method will run after the constructor and enforce that none of the parameters are null ('None' in Python parlance). Coding defensively this way will ensure that you values cannot change after initialization. Of course, you are free to add additional validations as well, such as acceptable values for the status of the category. However, should you wish to enforce values in that matter, it would be better to do that by making them proper classes or use the `Enum` class.

Next, let's review the `Hardware` class in the Mermaid diagram that we created in the last chapter.

Figure 4.4 The Hardware class that we drafted with the aid of ChatGPT. Note that it does not have any accessor methods

Hardware
<pre>+__init__(self, name: str, status: str, category: str, funding_details: FundingDetails, serial_number: str, purchase_date: str) -purchase_date: date -location: str -warranty_expiration_date: date -notes: str -maintenance_schedules: List[MaintenanceSchedule] -warranty: Warranty -retirement_date: date -retirement_reason: str -usage_statistics: UsageStatistics -budget: Budget</pre>

There are several items to note from this diagram: it contains neither accessor, nor mutator methods. We will need to add accessor methods to our prompt.

Now that we have reviewed and assessed what methods and attributes we need in our Hardware class, we should build it. Create a file called `hardware.py` in the domain directory. At the beginning of this file add the prompt in Listing 4.3.

Listing 4.3 A prompt to have GitHub Copilot create our Hardware class

```
# Define a class called Hardware, which is an instance of Asset.
# The Hardware entity has the following attributes:
# - serial_number: str,
# - location: Location,
# - warranty_expiration_date: date,
# - notes: str
# - maintenance_schedules: List[MaintenanceSchedule]
# - warranty: Warranty
# - retirement_date: date
# - retirement_reason: str
# - usage_statistics: UsageStatistics
# - budget: Budget
# The attributes can be None and the class should have a
constructor that takes all attributes as parameters.
```

```
# The attributes should be private and the class should have
accessor methods for all attributes.
```

The post constructor code and accessor methods are not included in code listing 4.4 for brevity's sake. However, they should be automatically added to your file.

Listing 4.4 The definition of the Hardware class

```
from datetime import datetime
from dataclasses import dataclass
from itam.domain.asset import Asset
from itam.domain.location import Location
from itam.domain.warranty import Warranty
from itam.domain.maintenance_schedule import MaintenanceSchedule
from itam.domain.usage_statistics import UsageStatistics
from itam.domain.budget import Budget

@dataclass
class Hardware(Asset):
    serial_number: str
    location: Location
    warranty_expiration_date: datetime
    notes: str
    maintenance_schedules: list[MaintenanceSchedule]
    warranty: Warranty
    retirement_date: datetime
    retirement_reason: str
    usage_statistics: UsageStatistics
    budget: Budget
```

Next, given that it was referenced in the Asset class, let's add the FundingDetails class next. First, however, let's review the class diagram. Notice one thing, the lines_of_business dictionary, which we will rename to department_allocations, uses a string as the key. We should change this to be an actual Department. This is another reminder that we must actively monitor the generated code and make decisions as to how to best implement the project.

Figure 4.5 The FundingDetails class that we drafted with the aid of ChatGPT. We will want to change the lines of business dict (renamed to department_allocations)

to use a Department as the key

FundingDetails
-name: str
-lines_of_business: Dict[str, float]
-depreciation_strategy: DepreciationStrategy
+__init__(self, name: str, lines_of_business: Dict[str, float], depreciation_strategy: DepreciationStrategy)
+get_name(self) : -> str
+get_lines_of_business(self) : -> Dict[str, float]
+get_line_of_business_percentage(self, line_of_business: str) : -> float
+get_depreciation_strategy(self) : -> DepreciationStrategy
+set_name(self, name: str)
+set_lines_of_business(self, lines_of_business: Dict[str, float])
+set_line_of_business_percentage(self, line_of_business: str, percentage: float)
+set_depreciation_strategy(self, depreciation_strategy: DepreciationStrategy)

We will want to make two additional edits to this class. We will want to include the Asset in the constructor. This will allow us access to the fields needed for calculating depreciations, as previously mentioned. Additionally, we will want to include a depreciation rate in the constructor. This is used for declining depreciation but not straight-line depreciation. We could set the value of this field in the specific depreciation constructor or using a mutator method, but for now, we will add it to the FundingDetails class. The updated prompt, including the updated field name, to create the FundingDetails class is listed in 4.5.

Listing 4.5 A prompt to have GitHub Copilot create our FundingDetails class

```
# Define a class called FundingDetails
# It should have the following attributes:
# - asset: Asset
# - depreciation_strategy: DepreciationStrategy,
# - depreciation_rate: float
# - department_allocations: Dict[Department, float]
# The attributes cannot be None and cannot be modified after
```

```
construction
# However, we should be able to access the attributes using
methods
```

There is nothing surprising within the generated class: the code establishes the internal attributes and sets them via the constructor. After the constructor has finished, it will check that all attributes are None.

Listing 4.6 The `FundingDetails` class, omitting the post constructor and the accessor methods

```
from dataclasses import dataclass
from item.domain.asset import Asset
from item.domain.depreciation_strategy import
DepreciationStrategy
from item.domain.department import Department

@dataclass
class FundingDetails:
    asset: Asset
    depreciation_strategy: DepreciationStrategy
    depreciation_rate: float
    department_allocations: Dict[Department, float]

    def __post_init__(self):
        if self.asset is None:
            raise TypeError("Asset cannot be None")
        if self.depreciation_strategy is None:
            raise TypeError("Depreciation strategy cannot be
None")
        if self.depreciation_rate is None:
            raise TypeError("Depreciation rate cannot be None")
        if self.department_allocations is None:
            raise TypeError("Department allocations cannot be
None")

    def get_asset(self):
        return self.asset

    def get_depreciation_strategy(self):
        return self.depreciation_strategy

    def get_depreciation_rate(self):
```

```
        return self.depreciation_rate

    def get_department_allocations(self):
        return self.department_allocations
```

The final class that we will generate in this chapter will be the `DepreciationStrategy`. In the last chapter we were briefly introduced to the `DepreciationStrategy` class. What was not mentioned at the time, is that this class would employ a new design pattern, the strategy pattern.

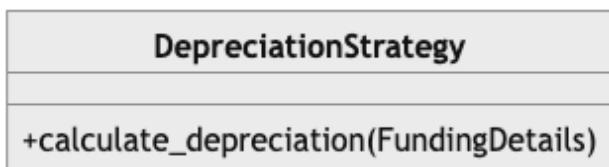
Strategy Design Pattern

The Strategy Design Pattern is a behavioral design pattern that lets you define a family of algorithms, encapsulate each one as an object, and make them interchangeable. The idea behind the Strategy Pattern is to define a common interface for a group of algorithms so that they're interchangeable, despite having potentially very different implementations.

4.1.4 Adapting a strategy for depreciation.

Before we attempt to create the `DepreciationStrategy` class, let's review the class diagram that we created last chapter.

Figure 4.6 The interface class `DepreciationStrategy` defines the functionality to calculate the depreciation of our assets within our domain model. There are four concrete classes representing how we can depreciate the asset.



The implementations of this class contain substantial hidden complexity. Without giving Copilot very specific instructions on how the calculations are to be performed, Copilot will not come up with the correct algorithms.

Listing 4.7 An incomplete and inexact prompt to get Copilot to create the DepreciationStrategies

```
# Define an interface called DepreciationStrategy.  
# It should have four concrete implementations of the interface:  
StraightLineDepreciationStrategy,  
DecliningBalanceDepreciationStrategy,  
DoubleDecliningDepreciationStrategy, and NoDepreciationStrategy.  
# Each implementation overrides the calculate_depreciation()  
method to provide a specific way of calculating depreciation for  
an asset based on its funding details.  
# The calculate_depreciation() method should take a  
FundingDetails object as a parameter and return a float value  
representing the depreciation amount.
```

This prompt will return methods that, at first sight, appear correct. However, they are not. For example, if we examine the straight-line depreciation method, we see that it neither includes the salvage value in the calculation nor uses the useful life.

Listing 4.8 An incorrect implementation of straight-line depreciation

```
class StraightLineDepreciationStrategy(DepreciationStrategy):  
    def calculate_depreciation(self, funding_details:  
        FundingDetails) -> float:  
        return funding_details.get_asset().get_cost() *  
            funding_details.get_depreciation_rate()
```

We should update the prompt with the actual formulas for the depreciation methods to correct this. Thankfully, we do not have to provide the exact syntax, but merely the calculations. Copilot will translate our algorithms to validate Python code. The updated prompt is in listing 4.9.

Listing 4.9 An accurate prompt includes the correct calculations for our depreciation methods.

```
# Define an interface called DepreciationStrategy.  
# It should have four concrete implementations of the interface:  
StraightLineDepreciationStrategy,  
DecliningBalanceDepreciationStrategy,
```

```

DoubleDecliningDepreciationStrategy, and NoDepreciationStrategy.
# Each implementation overrides the calculate_depreciation()
method to provide a specific way of calculating depreciation for
an asset based on its funding details.
# The calculate_depreciation() method should take a
FundingDetails object as a parameter and return a float value
representing the depreciation amount.
# NoDepreciationStrategy should return 0 for all assets.
# The other three strategies should return the depreciation
amount based on the following formulas:
# - Straight Line: (cost - salvage value) / useful_life
# - Declining Balance: cost * (1-rate/100)^(current_year -
purchase_year)
# - Double Declining: Declining Balance * 2

```

Note that we can use pseudocode. Copilot will handle the rest.

Listing 4.10 The output from our prompt featuring the correct calculates for our depreciation methods.

```

from abc import ABC, abstractmethod
from itam.domain.funding_details import FundingDetails
from math import pow

class DepreciationStrategy(ABC):
    @abstractmethod
    def calculate_depreciation(self, funding_details:
FundingDetails) -> float:
        pass

class StraightLineDepreciationStrategy(DepreciationStrategy):
    def calculate_depreciation(self, funding_details:
FundingDetails) -> float:
        cost = funding_details.get_asset().get_cost()
        salvage_value =
funding_details.get_asset().get_salvage_value()
        useful_life =
funding_details.get_asset().get_useful_life()
        return (cost - salvage_value) / useful_life

class
DecliningBalanceDepreciationStrategy(DepreciationStrategy):
    def calculate_depreciation(self, funding_details:
FundingDetails) -> float:

```

```

        cost = funding_details.get_asset().get_cost()
        depreciation_rate =
funding_details.get_depreciation_rate()
        return cost * pow(1 - (depreciation_rate / 100),
datetime.now().year - asset.get_purchase_date().year)

class DoubleDecliningDepreciationStrategy(DepreciationStrategy):
    def calculate_depreciation(self, funding_details:
FundingDetails) -> float:
        cost = funding_details.get_asset().get_cost()
        depreciation_rate =
funding_details.get_depreciation_rate()
        return
DecliningBalanceDepreciationStrategy().calculate_depreciation(fu
nding_details) * 2

class NoDepreciationStrategy(DepreciationStrategy):
    def calculate_depreciation(self, asset: Asset) -> float:
        return 0

```

Many other classes are left undefined. Therefore, you should feel free to practice perfecting your prompt engineering by implementing the missing classes. For example, you may want to add `Organizations`, `MaintenanceSchedule`, `Address`, `UsageStatistics`, `User`, `Role`, and `Warranty`.

In the next section, we will work with Copilot to apply patterns to our code, starting with a familiar one.

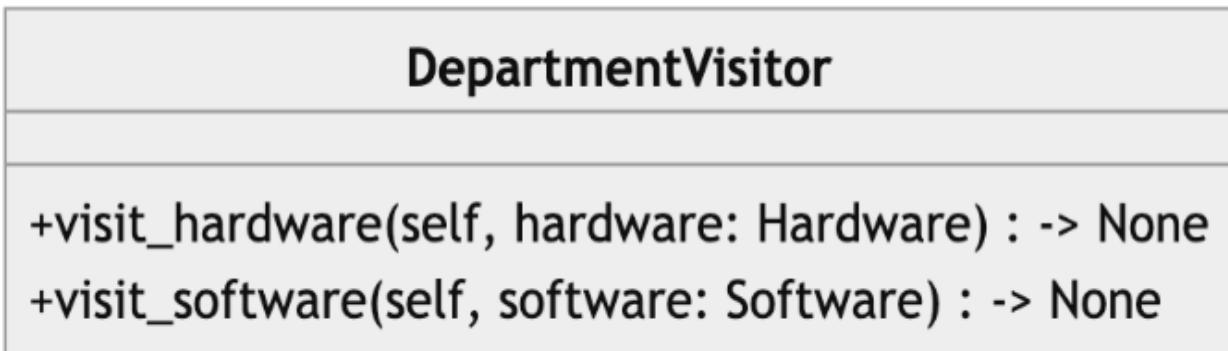
4.2 Weaving Patterns, Patterns, Patterns

In the following section, we will enrich our domain model with commonly used design patterns. Design patterns are common solutions to recurring design problems in software development. They represent best practices and can be used to speed up the development process by providing a set of proven solutions to certain problems. Design patterns are not ready-to-use code snippets but rather general guides on how to solve a particular problem in a way that is flexible and can be reused across different projects.

4.2.1 Paying a visit to our department

The first pattern that we will revisit is the *Visitor*. As you may recall from Chapter 3, you use the Visitor pattern to add new behavior to or change the existing behavior of a given class. First, let's look at how it was modeled in the last chapter.

Figure 4.7 The DepartmentVIitor interface (updated to use the new name, previously it was called LineOfBusinessVisitor). This class “visits” both hardware and software, holding the total cost and depreciation internally.



Let's design a prompt to have Copilot generate an implementation of our visitor pattern. `DepartmentVisitor`.

Listing 4.11 The prompt to have Copilot create a Vistor, which visits our Asset

```
# Define an interface called class DepartmentVisitor with the
# following methods:
# +visit(self, asset: Asset) -> None
# Additionally, define a class called class
# DepartmentStatisticsVisitor that implements the interface
# DepartmentVisitor
```

Copilot generates the straightforward implementation of this class.

Listing 4.12 The Vistor implementation

```
from abc import ABC, abstractmethod
from item.domain.asset import Asset

class DepartmentVisitor(ABC):
```

```

@abstractmethod
def visit(self, asset: Asset) -> None:
    pass

class DepartmentStatisticsVisitor(DepartmentVisitor):
    def __init__(self):
        self._total_cost = 0
        self._total_depreciation = 0
        self._total_allocation = 0

    def visit(self, asset: Asset) -> None:
        self._total_cost += asset.get_cost()
        self._total_depreciation += asset.get_depreciation()
        self._total_allocation += asset.get_allocation()

    def get_total_cost(self):
        return self._total_cost

    def get_total_depreciation(self):
        return self._total_depreciation

    def get_total_allocation(self):
        return self._total_allocation

```

4.2.2 Creating objects in a factory (pattern)

The *Factory* pattern is another creation pattern that allows you to abstract away some of the details of the object that you are attempting to create. In our Information Technology Asset Management system, we have two types of Assets (currently): Hardware and Software. Each of these objects have attributes that are idiosyncratic to themselves. One effect of this is that our application does not need to know the specifics of the concrete class (Hardware for example). From the application's perspective, it just gets an instance of Asset with additional fields. Let's have a look as to how to get Copilot to create a Factory for us.

Listing 4.13 The prompt to have Copilot create an AssetFactory class for us

```

# Define a class called AssetFactory
# It should have the following methods:
#     +new(self, asset_type: str, asset_name: str,

```

```

asset_cost: float, useful_life: int, depreciation_strategy: str,
depreciation_rate: float, salvage_value: float,
asset_purchase_date: str) -> Asset
# When asset_type is equal to "hardware" it should delegate the
call to a class called HardwareAssetFactory
# When asset_type is equal to "software" it should delegate the
call to a class called SoftwareAssetFactory
# If it is neither "hardware" nor "software" it should raise a
ValueError
# When depreciation_strategy is equal to "straight_line" it
should return a new instance of StraightLineDepreciationStrategy
# When depreciation_strategy is equal to
"double_declining_balance" it should return a new instance of
DoubleDecliningDepreciationStrategy
# When depreciation_strategy is equal to "none" it should return
a new instance of NoDepreciationStrategy
# When depreciation_strategy is equal to "declining_balance" it
should return a new instance of
DecliningBalanceDepreciationStrategy
# The HardwareAssetFactory and the SoftwareAssetFactory should
get the id from the AssetFactory on each call to new.

```

This prompt is one of the most detailed that we have encountered. As our needs get more and more niche, our prompts will naturally become more meticulous. We will examine this code in some detail, starting with the new method.

Listing 4.14 The resulting code for our AssetFactory class, containing two embedded Factories

```

from datetime import datetime
from itam.domain.asset import Asset
from itam.domain.hardware import Hardware
from itam.domain.funding_details import FundingDetails
from itam.domain.depreciation_strategy import (
    DepreciationStrategy,
    StraightLineDepreciationStrategy,
    DecliningBalanceDepreciationStrategy,
    DoubleDecliningDepreciationStrategy,
    NoDepreciationStrategy
)
class AssetFactory:

```

```

def __init__(self):
    self.id = 0

def date_from_string(self, date_string: str) -> datetime:
    return datetime.strptime(date_string, "%Y-%m-%d")

def new(self, asset_type: str, asset_name: str, asset_cost: float,
       useful_life: int, depreciation_strategy: str,
       depreciation_rate: float, salvage_value: float,
       asset_purchase_date: str) -> Asset:
    self.id += 1
    purchase_date =
        self.date_from_string(asset_purchase_date)

    if asset_type == "hardware":
        return HardwareAssetFactory().new(self.id,
                                           asset_name, asset_cost, useful_life,
                                           depreciation_strategy,
                                           depreciation_rate, salvage_value,
                                           purchase_date)
    elif asset_type == "software":
        return SoftwareAssetFactory().new(self.id,
                                           asset_name, asset_cost, useful_life,
                                           depreciation_strategy,
                                           depreciation_rate, salvage_value,
                                           purchase_date)
    else:
        raise ValueError

```

The new method will determine which of the specific Factories it will need to call, based upon the `asset_type` parameter. Additionally, it keeps track of the number of assets that it has created, ensuring that the identifier (“`id`” field) will allow be unique. Otherwise, if the identifier creation was pushed down to the specific Factory object, then we would get duplicate keys.

Note

This implementation is not thread safe. Access to the `id` fields should be synchronized.

Listing 4.15 The embedded `HardwareAssetFactory` class

```

class HardwareAssetFactory:
    def new(self, id: int, asset_name: str, asset_cost: float,
           useful_life: int, depreciation_strategy: str, depreciation_rate:

```

```

float, salvage_value: float, purchase_date: datetime) -> Asset:
    a = Hardware (
        id=id,
        name=asset_name,
        category="hardware",
        cost=asset_cost,
        useful_life=useful_life,
        status="active",
        salvage_value=salvage_value,
        purchase_date=purchase_date
    )

    funding_details=FundingDetails(
        asset=a,
        depreciation_strategy=

self._get_depreciation_strategy(depreciation_strateg,
        depreciation_rate=depreciation_rate,
        department_allocations=dict()
    )

    a.funding_details = funding_details
    return a

    def _get_depreciation_strategy(self, depreciation_strategy:
str) -> DepreciationStrategy:
        if depreciation_strategy == "straight_line":
            return StraightLineDepreciationStrategy()
        elif depreciation_strategy ==
"double_declining_balance":
            return DoubleDecliningDepreciationStrategy()
        elif depreciation_strategy == "none":
            return NoDepreciationStrategy()
        elif depreciation_strategy == "declining_balance":
            return DecliningBalanceDepreciationStrategy()
        else:
            raise ValueError

```

The new method of the `HardwareAssetFactory` class is relatively straightforward. This method accepts the parameters from the `AssetFactory` and attempts to resolve the `DepreciationStrategy`, and sets some sensible defaults.

Listing 4.16 The embedded `SoftwareAssetFactory` class

```

class SoftwareAssetFactory:
    def new(self, id: int, asset_name: str, asset_cost: float,
useful_life: int, depreciation_strategy: str, depreciation_rate:
float, salvage_value: float, purchase_date: datetime) -> Asset:
        a = Asset(
            id=id,
            name=asset_name,
            category="software",
            cost=asset_cost,
            useful_life=useful_life,
            status="active",
            salvage_value=salvage_value,
            purchase_date=purchase_date
        )

        funding_details=FundingDetails(
            asset=a,
            depreciation_strategy=self._get_depreciation_strategy(depreciation_strategy),
            depreciation_rate=depreciation_rate,
            department_allocations=dict()
        )

        a.funding_details = funding_details
        return a

    def _get_depreciation_strategy(self, depreciation_strategy: str) -> DepreciationStrategy:
        if depreciation_strategy == "straight_line":
            return StraightLineDepreciationStrategy()
        elif depreciation_strategy ==
"double_declining_balance":
            return DoubleDecliningDepreciationStrategy()
        elif depreciation_strategy == "none":
            return NoDepreciationStrategy()
        elif depreciation_strategy == "declining_balance":
            return DecliningBalanceDepreciationStrategy()
        else:
            raise ValueError

```

The SoftwareAssetFactory class is nearly identical to the HardwareAssetFactory class. So much so that it likely has a smell, and you

might find yourself with the urge to refactor, since this does appear to violate the DRY principle (Don't Repeat Yourself).

There is in fact an easier way to handle this de-duplication. In order to do this, we will look at our next design pattern: the *Builder* pattern.

The Builder Pattern

The Builder Pattern is a creational design pattern provides a fluent API for the creation of objects by providing step by step instructions on how to create said object.

4.2.3 Instructing the system on how to build

First, we will write a prompt to have Copilot create our builders: one for the Asset and one for the FundingDetails. We will have the Builder know that if the asset_type is hardware then it should return an instance of Hardware. The same for Software.

Listing 4.17 Prompt to create the AssetBuilder and the FundingDetailsBuilder classes

```
# Create a class called AssetBuilder
# It should use the Builder pattern to build an Asset
# Create another class called FundingDetailsBuilder
# It should use the Builder pattern to build a FundingDetails
# The AssetBuilder should have an embedded FundingDetailsBuilder
# When the category is "hardware" the AssetBuilder should create
# a Hardware object
# When the category is "software" the AssetBuilder should create
# a Software object
# When depreciation_strategy is equal to "straight_line" it
# should return a new instance of StraightLineDepreciationStrategy
# When depreciation_strategy is equal to
#"double_declining_balance" it should return a new instance of
DoubleDecliningDepreciationStrategy
# When depreciation_strategy is equal to "none" it should return
# a new instance of NoDepreciationStrategy
# When depreciation_strategy is equal to "declining_balance" it
```

```

should return a new instance of
DecliningBalanceDepreciationStrategy
# The variables will need to be held in local variables and then
passed to the Asset and FundingDetails objects when they are
created.
# The final method of the AssetBuilder should return an Asset
and be called build().
# The final method of the FundingDetailsBuilder should return a
FundingDetails and be called build().

```

One thing to note is that all of the values need to be stored in local variables; otherwise, we would encounter post initialization exceptions. The generated code for the AssetBuilder is in the following figure. The import statements and the accessor methods have been omitted for the sake of brevity.

Listing 4.18 The corresponding AssetBuilder classes

```

class AssetBuilder:
    def __init__(self):
        self.id = 0
        self.name = ""
        self.category = ""
        self.cost = 0.0
        self.useful_life = 0
        self.status = ""
        self.salvage_value = 0.0
        self.purchase_date = datetime.now()

    def with_name(self, name):
        self.name = name
        return self

    ...

```

Next, we will examine the `build()` method of the `AssetBuilder` class. This method will use the `category` field to return the correct which subclass of `Asset`.

Listing 4.19 The build() method for the AssetBuilder classes

```

def build(self) -> Asset:
    if self.category == "hardware":
        return Hardware(
            id=self.id,
            name=self.name,
            category=self.category,
            cost=self.cost,
            useful_life=self.useful_life,
            status=self.status,
            salvage_value=self.salvage_value,
            purchase_date=self.purchase_date
        )
    elif self.category == "software":
        return Software(
            id=self.id,
            name=self.name,
            category=self.category,
            cost=self.cost,
            useful_life=self.useful_life,
            status=self.status,
            salvage_value=self.salvage_value,
            purchase_date=self.purchase_date
        )
    else:
        return Asset(
            id=self.id,
            name=self.name,
            category=self.category,
            cost=self.cost,
            useful_life=self.useful_life,
            status=self.status,
            salvage_value=self.salvage_value,
            purchase_date=self.purchase_date
        )

```

We can now have a look at the `FundingDetailsBuilder`. This class will closely resemble the `AssetBuilder`, only without the polymorphic `build()` method.

Listing 4.20 The `FundingDetailsBuilder` classes

```

class FundingDetailsBuilder:
    def __init__(self):

```

```

        self.asset = None
        self.depreciation_strategy = ""
        self.depreciation_rate = 0.0
        self.department_allocations = dict()

    def with_asset(self, asset: Asset) -> FundingDetailsBuilder:
        self.asset = asset
        return self

    ...

```

The class's `build()` method implementation is fairly straightforward; it only returns an instance of the `FundingDetails` object after applying the parameters to the constructor.

Listing 4.21 The `build()` method of the `FundingDetailsBuilder` classes

```

def build(self) -> FundingDetails:
    return FundingDetails(
        asset=self.asset,
        depreciation_strategy=self.depreciation_strategy,
        depreciation_rate=self.depreciation_rate,
        department_allocations=self.department_allocations)

```

Next, let's extract the `_get_depreciation_strategy` method from the `AssetFactory` class, consolidating the logic of mapping the name of the depreciation strategy to an instance of the `DepreciationStrategy`.

Listing 4.22 The updated `build()` method of the `FundingDetailsBuilder` classes that includes the mapping from the str to the `DepreciationStrategy`

```

    def _get_depreciation_strategy(self, depreciation_strategy: str) -> DepreciationStrategy:
        if depreciation_strategy == "straight_line":
            return StraightLineDepreciationStrategy()
        elif depreciation_strategy == "double_declining_balance":
            return DoubleDecliningDepreciationStrategy()
        elif depreciation_strategy == "none":
            return NoDepreciationStrategy()
        elif depreciation_strategy == "declining_balance":

```

```

        return DecliningBalanceDepreciationStrategy()
    else:
        raise ValueError

    def build(self) -> FundingDetails:
        return FundingDetails(
            asset=self.asset,
            depreciation_strategy=self._get_depreciation_strategy(depreciation_strategy),
            depreciation_rate=self.depreciation_rate,
            department_allocations=self.department_allocations
        )

```

Now that we have written the Builders, we can modify the AssetFactory to use them.

A Pattern Hiding in Plain Sight: The Adapter

The Adapter Pattern is a structural design pattern that allows one to bridge the gap between the target interface and a class with an incompatible interface. For example, in our case, we could have formalized the interface for str -> DepreciationStrategy in an adapter called StringDepreciationStrategyAdapter.

We will update the prompt in the AssetFactory class to use the new Builders to construct the instances of Asset and FundingDetails.

Listing 4.23 The prompt to update the AssetFactory class to use the AssetBuilder

```

# Define a class called AssetFactory
# It should have the following methods:
#     +new(asset_type: str, asset_name: str, asset_cost:
float, useful_life: int, depreciation_strategy: str,
depreciation_rate: float, salvage_value: float) -> Asset
# Create a function that will take a string and return a
datetime
# Use the AssetBuilder to create the Asset and use the
FundingDetailsBuilder to create the FundingDetails

```

Using the new Builders will drastically reduce this class's code volume. For example, the new method is now only eight lines long (as formatted in Microsoft VS Code).

Listing 4.24 The prompt to update the AssetFactory class to use the AssetBuilder

```
def new(self, asset_type: str, asset_name: str, asset_cost: float, useful_life: int, depreciation_strategy: str, depreciation_rate: float, salvage_value: float, asset_purchase_date: str) -> Asset:
    self.id += 1
    purchase_date =
    self.date_from_string(asset_purchase_date)
    a =
    AssetBuilder().with_id(self.id).with_name(asset_name).with_categ
    ory(asset_type).with_cost(asset_cost).with_useful_life(useful_li
    fe).with_status("active").with_salvage_value(salvage_value).with
    _purchase_date(purchase_date).build()
    f =
    FundingDetailsBuilder().with_asset(a).with_depreciation_strategy
    (depreciation_strategy).with_depreciation_rate(depreciation_rate
    ).build()

    a.funding_details = f
    return a
```

4.2.4 Observing changes

Next, we are going to apply the Observer Pattern to our Information Technology Asset Management system. This pattern's real value will become more obvious in the next chapter, but we will sow the seeds here.

The Observer Pattern

The Observer Pattern is a behavioral pattern in which the Subject class reports certain state changes to Observer classes via notifications. The pattern is useful in User Interfaces and event-driven systems.

We will create a new class called AssetManager in the directory service. The AssetManager class will be responsible to interacting with the outside world (more on this in the next section). We will use the prompt in Listing 4.22 to instruct Copilot as to how to define this class.

Listing 4.25 A simple prompt to create an AssetManager

```
# Define a class called AssetManager  
# with CRUD operations for Asset
```

Our current implementation of AssetManager uses a list to track the lifecycle of the Assets that it manages. The AssetManager supports all of the CRUD (Create, Read, Update, and Delete) operations of Assets.

Listing 4.26 The implementation of the AssetManager

```
from itam.domain.asset import Asset

class AssetManager:  
    def __init__(self):  
        self._assets = []  
  
    def create(self, asset: Asset) -> None:  
        self._assets.append(asset)  
  
    def read(self, asset_id: int) -> Asset:  
        for asset in self._assets:  
            if asset.get_id() == asset_id:  
                return asset  
        raise ValueError("Asset not found")  
  
    def update(self, asset: Asset) -> None:  
        for i in range(len(self._assets)):  
            if self._assets[i].get_id() == asset.get_id():  
                self._assets[i] = asset  
                return  
        raise ValueError("Asset not found")  
  
    def delete(self, asset_id: int) -> None:  
        for i in range(len(self._assets)):  
            if self._assets[i].get_id() == asset_id:
```

```

        del self._assets[i]
    return
    raise ValueError("Asset not found")

def get_assets(self):
    return self._assets

```

We will update our prompt to add notifications to our AssetManager class. Right now, we will not assign any Observers. We will save that for next chapter, when we start to deal with data flow.

Listing 4.27 The updated prompt to apply the Observer Patter to the AssetManager

```

# Define a class called AssetManager
# with CRUD operations for Asset
# and a method to notify observers

```

In response to our prompt, if we recreate the entirety of the class, Copilot will add the notification framework on which we will eventually expand.

Listing 4.28 The updated AssetManager definition with notifications enabled

```

from itam.domain.asset import Asset

class AssetManager:
    def __init__(self):
        self._assets = []
        self.observers = []

    def add_observer(self, observer):
        self.observers.append(observer)

    def remove_observer(self, observer):
        self.observers.remove(observer)

    def notify_observers(self, event, asset):
        for observer in self.observers:
            observer.update(event, asset)

    def read(self, asset_id: int) -> Asset:
        for asset in self._assets:
            if asset.get_id() == asset_id:

```

```

        return asset
    raise ValueError("Asset not found")

def create(self, asset: Asset) -> None:
    self.notify_observers("CREATE", asset)
    self._assets.append(asset)

def update(self, asset: Asset) -> None:
    for i in range(len(self._assets)):
        if self._assets[i].get_id() == asset.get_id():
            self._assets[i] = asset
            self.notify_observers("UPDATE", asset)
            return
    raise ValueError("Asset not found")

def delete(self, asset_id: int) -> None:
    for i in range(len(self._assets)):
        if self._assets[i].get_id() == asset_id:
            del self._assets[i]
            self.notify_observers("DELETE", asset_id)
            return
    raise ValueError("Asset not found")

def get_assets(self):
    return self._assets

```

We will see two more patterns in the next section of this chapter: the Singleton Pattern and the Repository Pattern. However, before we see them, we will need to set up a Controller to interact with our system. That takes us to the usage of Ports and Adapters.

4.3 Plugging in Ports and Adapters

Our Information Technology Asset Management system is beginning to come together; the core business model has been built out. We have applied patterns to make the code more concise, readable, and maintainable. There is, however, one glaring deficiency: How do we interact with it? That is the subject of this section. In this section, we will continue to deepen our investigation into hexagonal architecture.

4.3.1 Hexagonal architecture in review

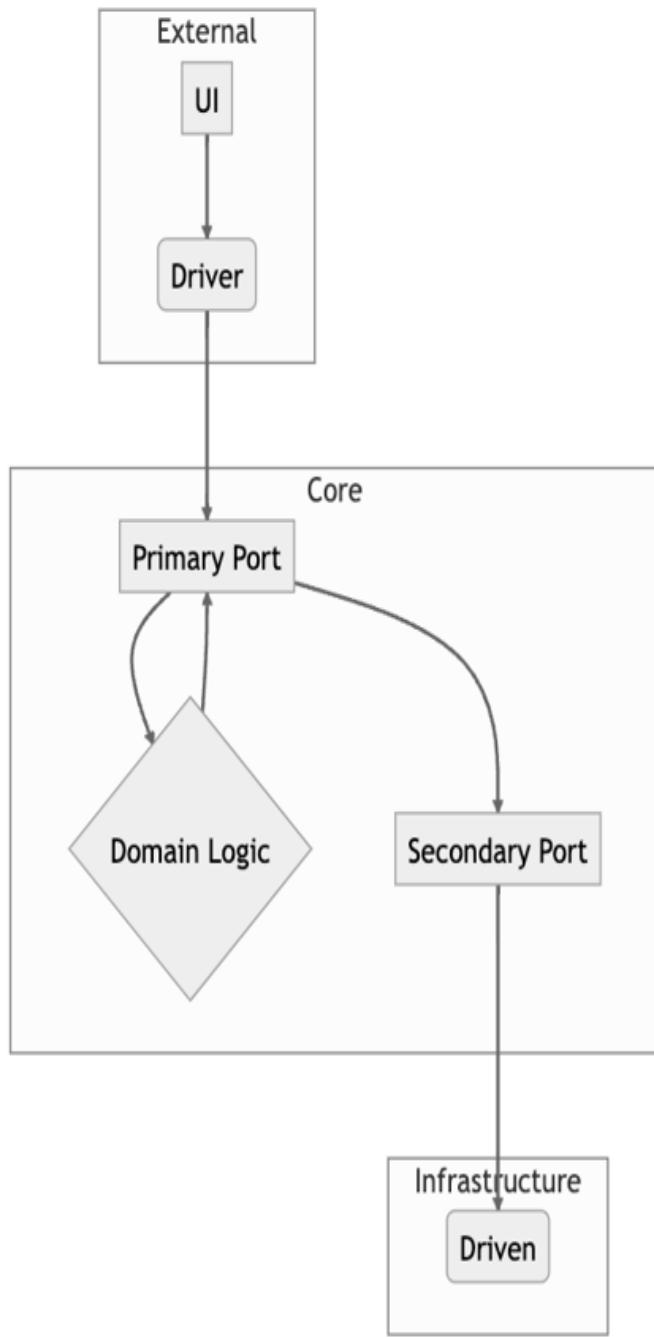
As you might recall hexagonal architecture is an approach to designing software that emphasizes the separation of the core business logic from the external services.

The business logic can be thought of as the "brain" of the application. It contains all the important rules and constructs that the application will need to guarantee the correctness of the program. In this analogy, then, the external services would be your "hands" or "eyes". They allow interactions with the outside world: user interfaces, database, etc.

Hexagonal architecture separates the main program logic from the outside parts like buttons, screens, and databases. It makes it easy to change those outside parts without changing the main program. It does this by using "ports" that define how the outside parts can interact with the main program, and "adapters" that make those interactions happen in a concrete way.

This approach makes it easier to change and evolve the application over time. If a change needs to be made to one of the external systems, the application's core should be unaffected. You would only need to update the adapter. The following diagram illustrates this:

Figure 4.8 A more conceptual visualization of hexagonal architecture in action. Notice that the Core is segregated from the rest of the system and is only interacted with via Ports.



4.3.2 Driving our application

We will begin by building out a driver of the system. A driver is a system external to the context boundary of the application that sends requests to the system and, optionally, receives responses from the application; a

conventional example being a REST call from a web browser to a REST controller.

First, we will add a REST controller to our Information Technology Asset Management system. The REST controller will expose the functionality provided by the `AssetManager` class. We will create a directory called `infrastructure/API`, in which we will create a file called `asset_controller.py`. At the beginning of this file add the following prompt.

Listing 4.29 The prompt to create a REST controller to expose the functionality of `AssetManager`

```
# Import statements for Asset, AssetService, AssetFactory,
AssetIn, AssetOut, and from_asset
# Use AssetFactory to create an Asset from an AssetIn
# Construct an instance of the APIRouter class from FastAPI
# Define HTTP endpoints to handle CRUD operations for the
AssetService
# Implementation of error handling, logging, and other necessary
components for a robust and reliable HTTP API
# All methods should be asynchronous
```

Given that we did not specify the path to use, it is possible that Copilot will produce inconsistent or invalid paths with the routes. This was what was output in VS Code, using the prompt above. More of a preference than a standard, but the URLs should be plural.

Listing 4.30 The generated code's route is “asset” rather than “assets”

```
@router.post('/asset', response_model=AssetOut)
async def create_asset(asset_in: AssetIn):
    asset = AssetFactory.create_asset(asset_in)
    asset_service.create(asset)
    return from_asset(asset)
```

You might elect to manually update the code to reflect your path preference. Above all, though, ensure that it is consistent across the various methods.

The first listing will show us how to initialize all of the services that we will need to run the application.

Listing 4.31 Updated AssetController code to reflect consistent routes

```
from fastapi import APIRouter, HTTPException
from itam.domain.asset import Asset
from itam.service.asset_manager import AssetManager
from itam.domain.factory.asset_factory import AssetFactory
from itam.infrastructure.api.asset_model import AssetIn,
AssetOut, from_asset
import logging

router = APIRouter()

asset_manager = AssetManager()
asset_factory = AssetFactory()
```

Next, we will see how the routes have been defined and their methods get used. The first two routes define how we will access our Asset objects. The first GET request will get all Assets that we currently have in the system. In the real world, we would include convenience features such as pagination and sorting. However, given the limited number of entries in the system currently, we shall forgo this. The next GET method will get a specific Asset by its identifier.

Listing 4.32 The AssetController methods to access Assets

```
@router.get('/assets', response_model=list[AssetOut])
async def get_assets():
    assets = asset_manager.get_assets()
    return [from_asset(asset) for asset in assets]

@router.get('/assets/{asset_id}', response_model=AssetOut)
async def read_asset(asset_id: int):
    try:
        asset = asset_manager.read(asset_id)
        return from_asset(asset)
    except ValueError as e:
        logging.error(e)
```

```
        raise HTTPException(status_code=404, detail="Asset not
found")
```

The final set of routes define how we will create, update, and remove Assets from our system. Note, that we are not doing “soft” deletes, which would just set a flag and this Asset would not be returned upon subsequent queries.

Listing 4.33 The AssetController methods to modify and delete Assets

```
@router.post('/assets', response_model=AssetOut)
async def create_asset(asset_in: AssetIn):
    asset = asset_factory.new(asset_in.asset_type,
asset_in.name, asset_in.unit_cost, asset_in.useful_life,
asset_in.depreciation_strategy, asset_in.depreciation_rate,
asset_in.salvage_value, asset_in.purchase_date)
    asset_manager.create(asset)
    return from_asset(asset)

@router.put('/assets/{asset_id}', response_model=AssetOut)
async def update_asset(asset_id: int, asset_in: AssetIn):
    try:
        asset = asset_factory.new(asset_in.asset_type,
asset_in.name, asset_in.unit_cost, asset_in.useful_life,
asset_in.depreciation_strategy, asset_in.depreciation_rate,
asset_in.salvage_value, asset_in.purchase_date)
        asset.set_id(asset_id)
        asset_manager.update(asset)
        return from_asset(asset)
    except ValueError as e:
        logging.error(e)
        raise HTTPException(status_code=404, detail="Asset not
found")

@router.delete('/assets/{asset_id}')
async def delete_asset(asset_id: int):
    try:
        asset_manager.delete(asset_id)
    except ValueError as e:
        logging.error(e)
        raise HTTPException(status_code=404, detail="Asset not
found")
```

You should note within that class that there are simplified representations of the Asset class called, not coincidentally, AssetIn and AssetOut. You could define those classes within the AssetController or you could add a separate class and define them there. In the reference implementation for this book, they are defined in a file called asset_model.py.

Listing 4.34 The prompt to define the AssetIn and AssetOut classes

```
# Define a pydantic model for AssetIn
# It should have the following attributes:
#     name: str
#     asset_type: str
#     department: str
#     depreciation_strategy: str
#     useful_life: int
#     unit_cost: float
#     depreciation_rate: float
#     salvage_value: float

# Define a pydantic model for AssetOut
# It should have the following attributes:
#     id: int
#     name: str
#     asset_type: str
#     department: str
#     depreciation_strategy: str
#     useful_life: int
#     unit_cost: float
#     depreciation_rate: float
#     salvage_value: float
# It should have a method that transforms an Asset into an
AssetOut
```

With the prompt defined at the beginning of the file, we should get code generated by Copilot that is similar to the code in listing 4.35.

Listing 4.35 The classes AssetIn and AssetOut in asset_model.py

```
from pydantic import BaseModel
from itam.domain.asset import Asset
```

```

class AssetIn(BaseModel):
    name: str
    asset_type: str
    department: str
    depreciation_strategy: str
    useful_life: int
    unit_cost: float
    depreciation_rate: float
    salvage_value: float
    purchase_date: str

class AssetOut(BaseModel):
    id: int
    name: str
    asset_type: str
    depreciation_strategy: str
    useful_life: int
    unit_cost: float
    depreciation_rate: float
    salvage_value: float
    purchase_date: str

@staticmethod
def from_asset(asset: Asset) -> AssetOut:
    return AssetOut(
        id=asset.id,
        name=asset.name,
        asset_type=asset.category,
        depreciation_strategy=str(asset.funding_details.depreciation_strategy),
        useful_life=asset.useful_life,
        unit_cost=asset.cost,
        depreciation_rate=asset.funding_details.depreciation_rate,
        salvage_value=asset.salvage_value,
        purchase_date=asset.purchase_date.strftime("%Y-%m-%d"),
    )

```

We could, optionally, add another layer of abstraction between the AssetController and the AssetManager. One might consider during so might be hexagonal architecture in extremis, but there are few challenges in Computer Science that cannot be resolved by adding one more layer of abstraction. We will add a Port that will be used to interface with the

`AssetController`. Let's create a prompt for the class `AssetRestPort` and have Copilot generate the code.

Listing 4.36 A prompt to create the `AssetRestPort`

```
# Define an interface called AssetRestPort
# It should expose all of the methods in AssetManager as abstract
methods
```

This prompt instructs Copilot to create an interface that we will implement shortly.

Listing 4.37 Complete source code for `AssetRestPort`

```
from itam.domain.asset import Asset
from abc import abstractmethod

class AssetRestPort:
    @abstractmethod
    def read(self, asset_id: int) -> Asset:
        pass

    @abstractmethod
    def create(self, asset: Asset) -> None:
        pass

    @abstractmethod
    def update(self, asset: Asset) -> None:
        pass

    @abstractmethod
    def delete(self, asset_id: int) -> None:
        pass

    @abstractmethod
    def get_assets(self):
        pass
```

Next, we will implement this Port with an Adapter that will be used to interact with the Controller to further abstract the components. We would be able to swap out this Adapter with another, should we want to turn this

into a command line application, for example. The prompt for the Adapter is as follows.

Listing 4.38 A prompt to create the AssetRestAdapter

```
# Define an interface called AssetRestAdapter
# It should take an AssetManager as a parameter in its
constructor
# It should expose all of the methods in AssetManager
# It should inherit from AssetRestPort
```

There are two important elements to this prompt. The first is that it implements the Port interface that we had previously defined. The second is that it takes the AssetManager and wraps its functionality.

Listing 4.39 The source code for AssetRestAdapter

```
from item.domain.asset import Asset
from item.infrastructure.ports.asset_rest_port import
AssetRestPort
from item.service.asset_manager import AssetManager

class AssetRestAdapter(AssetRestPort):
    def __init__(self, asset_manager: AssetManager):
        self._asset_manager = asset_manager

    def read(self, asset_id: int) -> Asset:
        return self._asset_manager.read(asset_id)

    def create(self, asset: Asset) -> None:
        self._asset_manager.create(asset)

    def update(self, asset: Asset) -> None:
        self._asset_manager.update(asset)

    def delete(self, asset_id: int) -> None:
        self._asset_manager.delete(asset_id)

    def get_assets(self):
        return self._asset_manager.get_assets()
```

All that is left to do is to update the `AssetController` to remove the direct invocation of the `AssetManager` methods by the `AssetController`, and rather have the `AssetController` invoke the methods of the Adapter, which in turn invokes the methods of the `AssetManager`. The key takeaway from the Ports and Adapters pattern is that it abstracts the interaction between the driving parts of the system, in this case the REST API, and the driven application: the business model and core of our system: the `AssetManager`. To make this more explicit and as a little preview, we will shortly modify this class yet again to add the Port to the constructor.

Listing 4.40 Updated code for `AssetController` which uses the `AssetRestAdapter`

```
router = APIRouter()

asset_rest_adapter = AssetRestAdapter(AssetManager())
asset_factory = AssetFactory()

@router.post('/assets', response_model=AssetOut)
async def create_asset(asset_in: AssetIn):
    asset = asset_factory.new(asset_in.asset_type,
    asset_in.name, asset_in.unit_cost, asset_in.useful_life,
    asset_in.depreciation_strategy, asset_in.depreciation_rate,
    asset_in.salvage_value, asset_in.purchase_date)
    asset_rest_adapter.create(asset)
    return from_asset(asset)
```

As previously mentioned, we will modify the `AssetController` to remove all direct references to the `AssetManager`. Our current `AssetController`, while it does not directly invoke any `AssetManager` methods, does hold an indirect reference to the `AssetManger`, as the `AssetManager` is constructed in the `AssetController`. Doing so will further insulate the `AssetManager` from the changes of the drivers.

4.3.3 Accessing our data and persisting our changes

Abstractions in software are not merely an academic matter. These abstractions allow one flexibility in how one can satisfy the requirements and make changes easier to make. They can hide complexity, encapsulating

behavior behind seemingly simple interfaces. This will bring us to our next pattern: the *Repository Pattern*.

The Repository Pattern

The Repository pattern is a design pattern that helps manage data by creating a separate place to handle tasks like getting, saving, updating, and deleting data. It connects the data source (like a database or file) to the rest of the application, keeping data access code apart from other code.

In our ITAM project, the Repository Pattern will allow us to present a greatly simplified interface, masking the complexity of, in this case, our data access. From the perspective of the Repository or the AssetManager for that matter, our data access layer is just a series of methods. Behind the scenes it might be a REST API, a Redis cache, MongoDB, a file, or a database of any flavor. Let's create a prompt that will set up our generic data access layer, using the Repository Pattern.

Listing 4.41 The prompt to create our generic data access layer

```
# Create an abstract base class called BaseRepository that
# represents a generic data access layer using the Repository
# pattern.
# The class should define common methods for CRUD operations
# such as create, read, update, and delete.
# Make sure to use type hints for the input parameters and
# return values.
```

With this prompt, Copilot should have everything it needs to establish the contract by which all implementations will honor and behave. Let's examine the code.

Listing 4.42 The code for the BaseRepository which will function as our generic data access layer

```
from abc import ABC, abstractmethod
from typing import List, TypeVar, Generic
```

```

T = TypeVar('T')

class BaseRepository(ABC, Generic[T]):
    @abstractmethod
    def create(self, entity: T) -> None:
        pass

    @abstractmethod
    def read(self, entity_id: int) -> T:
        pass

    @abstractmethod
    def update(self, entity: T) -> None:
        pass

    @abstractmethod
    def delete(self, entity_id: int) -> None:
        pass

    def get_entities(self) -> List[T]:
        pass

```

Copilot introduced a Generic type, allowing for some typing hints when it comes time to implement. If we express that the type should be an Asset, then the method signature would demonstrate this. Our first attempt at an implementation will be an in-memory data store. This will allow us to perform an end-to-end test of all of our system. It also nicely demonstrates why abstractions are useful. Before long, we will add another implementation that uses Postgres. But first, the prompt:

Listing 4.43 The prompt to an in-memory data store for testing purposes

```

# Define a class called InMemoryAssetRepository that inherits
from the BaseRepository abstract class.
# The InMemoryAssetRepository should be specific to managing
Asset objects sing an in-memory list to store and update the
assets.
# Implement the necessary methods for CRUD operations such as
create, read, update, and delete for Asset objects using the in-
memory list.

```

```
# Make sure to use type hints for the input parameters and
return values.
```

Our in-memory implementation will use a simple list in which to store the Assets. We will be able to perform all CRUD operations. Additionally, Copilot will add in the typing hints for us: letting others know that it should accept and return Assets. That is what is meant by `BaseRepository[Asset]`: this is an implementation of a `BaseRepository` whose Generic Type is `Asset`.

Listing 4.44 The code for the `InMemoryAssetRepository`

```
from itam.infrastructure.repository.base_repository import
BaseRepository
from itam.domain.asset import Asset

class InMemoryAssetRepository(BaseRepository[Asset]):
    def __init__(self):
        self._assets = []

    def create(self, asset: Asset) -> None:
        self._assets.append(asset)

    def read(self, asset_id: int) -> Asset:
        return next((asset for asset in self._assets if asset.id
== asset_id), None)

    def update(self, asset: Asset) -> None:
        for i in range(len(self._assets)):
            if self._assets[i].id == asset.id:
                self._assets[i] = asset
                break

    def delete(self, asset_id: int) -> None:
        self._assets = [asset for asset in self._assets if
asset.id != asset_id]

    def get_entities(self) -> list[Asset]:
        return self._assets
```

Finally, we will update the `AssetManager` to delegate the CRUD operations of the Assets to a `BaseRepository` instance (`_repository`). The full source

code is below, including the prompt, located at the beginning of the file.

Listing 4.45 The updated prompt and code for the AssetManager to use the InMemoryAssetRepository

```
# Define a class called AssetManager
# with CRUD operations for Asset
# and a method to notify observers whenever an asset is created,
updated or deleted
# The AssetManager should use an instance of the
InMemoryAssetRepository class for data access and CRUD
operations on Asset objects.
# Implement methods for creating, reading, updating, and
deleting assets using the AssetRepository instance.
# Please include type hints for the input parameters and return
values.
# The methods should be named create, read, update, get_assets,
and delete.

from item.domain.asset import Asset
from item.infrastructure.repository.in_memory_asset_repository
import InMemoryAssetRepository

class AssetManager:
    def __init__(self):
        self._repository = InMemoryAssetRepository()

    def create(self, asset: Asset) -> Asset:
        self._repository.create(asset)
        return asset

    def read(self, asset_id: int) -> Asset:
        return self._repository.read(asset_id)

    def update(self, asset: Asset) -> Asset:
        self._repository.update(asset)
        return asset

    def get_assets(self) -> list[Asset]:
        return self._repository.get_entities()

    def delete(self, asset_id: int) -> None:
        self._repository.delete(asset_id)
```

At this point, we have a core business domain that is untouched directly by our system. We have ports by which requests can come in. We also have ports by which we can store the data (at least while the system is running). We should be able to test the system end-to-end, by running it and POST-ing request to the creation end point, and the reading from the GET end points. Once we have confirmed that the system works end-to-end, we can now tackle the issue that our data is only persisted in memory; we can now hook up an actual database. In order to do this, we introduce the final pattern of this chapter: the *Singleton*. The Singleton, conceptually, is very simple to grok; there should only ever be one instance of it running. The Singleton is good for many use cases: logging, caching, configuration settings, or a database connection manager.

The Singleton Pattern

The Singleton pattern is a design pattern that ensures a class has only one instance and provides a global access point to that instance. It is used when you want to have a single object shared across different parts of your program, instead of creating multiple instances of the same class.

4.3.4 Centralizing (and externalizing) our data access

We will ask Copilot to create a Singleton class for us to manage our database's connections. Since we never want to hard code usernames or passwords (or any connection details for that matter) in our source code, not only because it is inherently less secure, but the details will likely differ depending on the environment (DEV versus QA versus PROD). We will therefore ask Copilot to accept these values as environmental variables and configure the connections using these values.

Listing 4.46 A prompt to create the Singleton DatabaseConnection class

```
# Create a Python class called DatabaseConnection using the
# Singleton pattern to manage a single connection to a PostgreSQL
# database using SQLAlchemy.
# The class should read the database username, password, and
```

```
connection string from environment variables
# The environment variables should be named: DB_USERNAME,
DB_PASSWORD, DB_HOST, DB_PORT, and DB_NAME
```

We have asked Copilot to employ SQLAlchemy, an Object Relational Mapping (ORM) tool, to perform the actions against the database. Copilot will adroitly assemble our source code. The `@staticmethod` keyword will create a method that belongs to class rather than the instance of the class. This keyword is used to get the instance of the `DatabaseConnection` class. Since a static method cannot modify the instance data, it can be used to ensure that only one instance of this class is ever running, and thus is a Singleton. The constructor initializes the connection to the database using string interpolation using environment variables.

Listing 4.47 A prompt to create the Singleton `DatabaseConnection` class

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
import os

class DatabaseConnection:
    __instance = None
    @staticmethod
    def get_instance():
        if DatabaseConnection.__instance == None:
            DatabaseConnection()
        return DatabaseConnection.__instance

    def __init__(self):
        if DatabaseConnection.__instance != None:
            raise Exception("This class is a singleton!")
        else:
            DatabaseConnection.__instance = self
            self.engine =
create_engine(f"postgresql://{{os.environ['DB_USERNAME']}":
{{os.environ['DB_PASSWORD']}@{{os.environ['DB_HOST']}":
{{os.environ['DB_PORT']}/{{os.environ['DB_NAME']}}}")
            self.Session = sessionmaker(bind=self.engine)

    def get_session(self):
        return self.Session()
```

Now that we have a connection to our database, we will need to create a repository that will use this connection to store and retrieve our Asset objects. We will create a class called `SQLAlchemyAssetRepository`. This name very explicitly states what it is and how it does it.

Listing 4.48 A prompt and code to the `SQLAlchemyAssetRepository` class

```
# Define a class called SQLAlchemyAssetRepository that
# implements the AssetRepository interface for data access using
# SQLAlchemy.
# The class should handle the CRUD operations (create, read,
# update, and delete) for assets, storing and retrieving them in a
# PostgreSQL database using SQLAlchemy.

from item.domain.asset import Asset
from item.infrastructure.repository.base_repository import
BaseRepository
from sqlalchemy.orm import Session

class SQLAlchemyAssetRepository(BaseRepository[Asset]):
    def __init__(self, session: Session):
        self._session = session

    def create(self, asset: Asset) -> Asset:
        self._session.add(asset)
        self._session.commit()
        return asset

    def read(self, asset_id: int) -> Asset:
        return self._session.query(Asset).filter(Asset.id ==
asset_id).first()

    def update(self, asset: Asset) -> Asset:
        self._session.query(Asset).filter(Asset.id ==
asset.id).update(asset)
        self._session.commit()
        return asset

    def get_assets(self) -> list[Asset]:
        return self._session.query(Asset).all()

    def delete(self, asset_id: int) -> None:
        self._session.query(Asset).filter(Asset.id ==
```

```
asset_id).delete()
    self._session.commit()
```

Next, we will refactor the AssetController to allow us to pass in the AssetRestPort. By doing this, we can swap out the driven aspect of our application. This refactor will further abstract the driver from the business core.

Listing 4.49 The updated AssetController class which exposes the routes rather than using the method decorators

```
from itam.infrastructure.ports.asset_rest_port import
AssetRestPort
import logging

class AssetController:
    def __init__(self, asset_rest_port: AssetRestPort):
        self._asset_factory = AssetFactory()
        self._asset_rest_port = asset_rest_port
        ...

    def get_router(self):
        return self._router

    @asyncio.coroutine
    def get_assets(self):
        return [from_asset(a) for a in
self._asset_rest_port.get_assets()]

    @asyncio.coroutine
    def get_asset(self, asset_id: int):
        asset = self._asset_rest_port.read(asset_id)
        if asset is None:
            raise HTTPException(status_code=404, detail="Asset
not found")
        return from_asset(asset)

    @asyncio.coroutine
    def create_asset(self, asset_in: AssetIn):
        asset = self._asset_factory.new(asset_in.asset_type,
asset_in.name, asset_in.unit_cost, asset_in.useful_life,
asset_in.depreciation_strategy, asset_in.depreciation_rate,
asset_in.salvage_value, asset_in.purchase_date)
        self._asset_rest_port.create(asset)
        return from_asset(asset)
```

```

    async def update_asset(self, asset_id: int, asset_in:
AssetIn):
        asset = self._asset_factory.new(asset_in.asset_type,
asset_in.name, asset_in.unit_cost, asset_in.useful_life,
asset_in.depreciation_strategy, asset_in.depreciation_rate,
asset_in.salvage_value, asset_in.purchase_date)

        asset.id = asset_id
        asset = self._asset_rest_port.update(asset)
        if asset is None:
            raise HTTPException(status_code=404, detail="Asset
not found")
        return from_asset(asset)

    async def delete_asset(self, asset_id: int):
        asset = self._asset_rest_port.read(asset_id)
        if asset is None:
            raise HTTPException(status_code=404, detail="Asset
not found")
        self._asset_rest_port.delete(asset_id)
        return from_asset(asset)

```

We can now consolidate the initialization logic of our application into the `main.py` file. This is the big payoff. Our system will have layering, facilitating the swapping out of the components are needed or as requirements change.

Listing 4.50 The finalized version of the `main.py` class, in which we wire our application together

```

from fastapi import FastAPI
from itemam.infrastructure.api.asset_controller import
AssetController
#from itemam.infrastructure.repository.in_memory_asset_repository
import InMemoryAssetRepository
from itemam.infrastructure.repository.sqlalchemy_asset_repository
import SQLAlchemyAssetRepository
from itemam.infrastructure.database.database_connection import
DatabaseConnection
from itemam.service.asset_manager import AssetManager
from itemam.infrastructure.adapters.asset_rest_adapter import
AssetRestAdapter
import uvicorn

```

```

app = FastAPI()
session = DatabaseConnection().get_session()
#repository = InMemoryAssetRepository()
repository = SQLAlchemyAssetRepository(session)
asset_manager = AssetManager(repository)
asset_rest_adapter = AssetRestAdapter(asset_manager)
asset_controller = AssetController(asset_rest_adapter)
app.include_router(asset_controller.get_router())

if __name__ == '__main__':
    uvicorn.run(app, host='0.0.0.0', port=8000)

```

Congratulations, we now have a running system, which is persisting data to our database.

4.4 Summary

- The Decorator Design Pattern is a structural design pattern that allows you to add new or modify existing behavior to an object dynamically without changing the existing class. This is done by wrapping the current object in the decorating object.
- Visitor pattern to add new behavior to or change the existing behavior of a given class.
- The Factory pattern is another creation pattern that allows you to abstract away some of the details of the object that you are attempting to create.
- The Builder Pattern is a creational design pattern provides a fluent API for the creation of objects by providing step by step instructions on how to create said object.
- The Adapter Pattern is a structural design pattern that allows one to bridge the gap between the target interface and a class with an incompatible interface.
- The Observer Pattern is a behavioral pattern in which the Subject class reports certain state changes to Observer classes via notifications.
- Hexagonal architecture separates the main program logic from the outer parts like buttons, screens, and databases. It makes changing outer parts easy without changing the main program.

- The Repository pattern is a design pattern that helps manage data by creating a separate place to handle tasks like getting, saving, updating, and deleting data. It connects the data source (like a database or file) to the rest of the application, keeping data access code apart from other code.
- The Singleton pattern is a design pattern that ensures a class has only one instance and provides a global access point to that instance. It is used when you want to have a single object shared across different parts of your program instead of creating multiple instances of the same class.

5 Managing Data with GitHub Copilot and Copilot Chat

This chapter covers

- Persisting our data into a relational database
- Streaming our data using Apache Kafka
- Incorporating event-driven principles
- Analyzing our data to monitor the location using Spark

The last chapter laid the foundation for our Information Technology Asset Management system. However, this application will not fulfill our requirements without data. Data is the life's blood of every application. That is what this chapter is all about: the various ways that we can use Generative AIs to create data, stream data, transform data, react to data, and learn from data.

Perceptive individuals might have noticed in the last chapter that our data access pattern would not have worked as it was incomplete. The opening section of this chapter will address this. After that, we will set up our database, fix the classes which access this data, and load some sample data for us to use in the rest of the chapter.

5.1 Amassing our data set

Our first task will be to construct a substantial corpus of data to assist our experimentation in the remainder of the chapter. First, we will use GitHub Copilot to generate one thousand rows of asset information. We will soon find, however, that this may not be the tool most suited to this task. One key driver behind using these tools is the idea of discovery: Testing their boundaries, pushing against them, and occasionally, pushing back. But the journey is often where the joy is found. Once we have found this edge, we will be introduced to a new, previously unseen tool: GitHub Copilot Chat.

Finally, once we have created our list of assets, we will add location information for those assets, again using GitHub Copilot Chat.

We need to get our database running before building our initial dataset. Docker makes this task trivial, allowing us to quickly spin up an empty Postgres (or other RDBMS/NoSQL server) with minimal effort. Have you forgotten the command to do this? No worries, we can ask Copilot. Open a new file called data/initial_data_laod.sql. and enter the following prompt at the top of your newly minted SQL file.

Snippet 5.1 A prompt to have GitHub Copilot provide us with a Docker command

```
-- Question: What is the command to run a docker container,  
running PostgreSQL for a database called item_db. I would like  
to specify the password for this database.
```

Copilot will slowly reveal the Docker command: --Answer: docker run --name item_db -e POSTGRES_PASSWORD=postgres -d -p 5432:5432 postgres. Once you run this command at your terminal or command line, we can build out our dataset. You should be able to connect to the locally running database. You should notice that there is a database called item_db running in it. However, this database has no schema, tables, or data. Let's first set up a new schema.

In our initial_data_file.sql, we will add a prompt to have Copilot draft the schema creation command. The following prompt (and response from Copilot) will allow you to create a new schema called item if executed from within your database client application (e.g., DataGrip, SQuirreL, pdAdmin, or even using the Docker exec command docker exec -i item_db psql -U postgres -c "create schema item")

Snippet 5.2 A prompt to have Copilot create a new schema

```
-- create a schema called item  
create schema item;
```

Next, we should add a user for use within our application. This user will be able to perform *CRUD* (Create, Read, Update, Delete) operations on our data but will not be able to affect the structure of the database tables or procedures.

Note

The lines that start with double dashes are comments in SQL. Commenting out these lines are optional from Copilot's perspective, as it will generate solutions without the comments; it makes it easier to copy and paste the code directly into our database tool of choice.

While we are at it, we will also add an administrative account to perform the operations that our read-write users cannot, such as creating or dropping tables.

Listing 5.1 A prompt to create new users

```
-- create two users called item_user and item_admin. item_user  
should be able to read from and write to the tables in the item  
schema. item_admin should be able to read from and write to the  
tables in the item schema and create and drop tables in the item  
schema.  
  
create user item_user with password 'item_user';  
create user item_admin with password 'item_admin';  
grant usage on schema item to item_user;  
grant usage on schema item to item_admin;  
grant select, insert, update, delete on all tables in schema  
item to item_user;  
grant select, insert, update, delete on all tables in schema  
item to item_admin;  
grant usage, drop on schema item to item_admin;
```

Next, we will grant ownership of this schema to the item_admin account. Transferring this ownership will ensure that only this account can change the table structure: the data definition.

Snippet 5.3 A prompt to transfer schema ownership to the admin account

```
-- grant ownership of the item schema to item_admin  
alter schema item owner to item_admin;
```

With the set-up, account creation, and worship of the system out of the way, we can start to focus on the data. We will begin by adding the reference data, the data that supports the assets: the depreciation strategies. This data is more static in nature; it changes less frequently, if at all. Next, we will define and store these strategies.

Listing 5.2 A prompt to create the depreciation_strategy table

```
-- create a table called depreciation_strategy in the item  
schema. the table should have the following columns: id (int),  
name (varchar), and description (varchar). the table should have  
a primary key on id.  
-- id needs to be in quotes because it is a reserved word in  
postgresql  
-- there are two values for depreciation_strategy: straight line  
and double declining balance  
create table item.depreciation_strategy (  
    "id" int primary key,  
    "name" varchar,  
    "description" varchar  
);
```

We will use a sequence as this table's primary key. While this would not be strictly necessary for a table that will not be very large and with known values that we could and will manually enter, adding this sequence will allow us to work with Copilot more and have it make some suggestions. Moreover, it is amusing to ask Copilot questions and have Copilot answer within a text file.

Listing 5.3 A prompt to create a sequence for use as the primary key of the depreciation_strategy table

```
-- create a sequence called depreciation_strategy_seq, which  
should start at 1 and increment by 1 and should be used as the  
primary key for the depreciation_strategy table.  
create sequence item.depreciation_strategy_seq start 1 increment  
1;
```

Naturally, with the sequence in our proverbial hand, we need to know how to associate the sequence with the primary key column of the depreciation_strategy table. Luckily, Copilot has the answer.

Listing 5.4 Asking Copilot how to associate the sequence with the primary key

```
-- question: how do I make the sequence the primary key for the  
depreciation_strategy table?  
  
-- answer: use the following command  
alter table item.depreciation_strategy alter column "id" set  
default nextval('item.depreciation_strategy_seq'::regclass);
```

Finally, we shall complete this table by inserting the following static entries into the table. We will only use two depreciation strategies for now: straight-line and double declining balance.

Listing 5.5 Adding the static entries to the depreciation_strategy table

```
insert into depreciation_strategy (id, name, description) values  
(1, 'straight line', 'straight line');  
  
insert into depreciation_strategy (id, name, description) values  
(2, 'double declining balance', 'double declining balance');
```

Next, we shall move on the funding_details table. This information tells us how we financed our equipment, the resale value, and instructions for what should be done with our asset once its useful life is over. The sequence of steps will be identical for what we did for the depreciation strategies, with the exception that we will not be added static entries, as this data is directly related to an individual asset. We will define the table, create the sequence, and apply said sequence to the table, functioning as the primary key.

Listing 5.6 Complete code listing for the funding_details table

```
-- create a table called funding_details in the item schema. the  
table should have the following columns: id (int), name  
(varchar), depreciation_strategy_id (int) and depreciation_rate
```

```

(float). the table should have a primary key on id.
-- depreciation_strategy_id is a foreign key to the
depreciation_strategy table.
-- id needs to be in quotes because it is a reserved word in
postgresql
create table item.funding_details (
    "id" int primary key,
    "name" varchar,
    "depreciation_strategy_id" int,
    "depreciation_rate" float
);
-- create a sequence called funding_details_seq, which should
start at 1 and increment by 1 and should be used as the primary
key for the funding_details table.
create sequence item.funding_details_seq start 1 increment 1;

alter table item.funding_details alter column "id" set default
nextval('item.funding_details_seq'::regclass);

```

The final information that we will define and generate are the assets themselves. This listing, too, is redundant but included for completeness. Finally, we create the table, make the sequence, and use it as the primary key.

Listing 5.7 Complete code listing for the assets table

```

-- create a table called assets in the item schema. the table
should have the following columns:
-- id (int), name (varchar), status (varchar), category
(varchar), cost (float), useful_life (int), salvage_value
(float), purchase_date (date), funding_details_id (int). The
table should have a primary key on id and a foreign key on
funding_details_id.
-- id needs to be in quotes because it is a reserved word in
postgresql
-- the table should have a sequence called assets_id_seq, which
should start at 1 and increment by 1 and should be used as the
primary key for the assets table.
create table item.assets (
    "id" int primary key,
    "name" varchar,
    "status" varchar,

```

```

    "category" varchar,
    "cost" float,
    "useful_life" int,
    "salvage_value" float,
    "purchase_date" date,
    "funding_details_id" int
);

-- create a sequence called assets_seq, which should start at 1
and increment by 1 and should be used as the primary key for the
assets table.
create sequence itam.assets_seq start 1 increment 1;

alter table itam.assets alter column "id" set default
nextval('itam.assets_seq'::regclass);

```

With the tables defined and created, we will now focus on creating the data. In our text file, we instruct Copilot with parameters for the dataset we are looking for. Copilot will likely attempt to assist you in outlining the attributes surrounding your new dataset.

Listing 5.8 Creating a dataset for the assets table

```

-- Generate a dataset of assets for an ITAM system. The dataset
should include the following columns: id (int), name (varchar),
status (varchar), category (varchar), cost (float), useful_life
(int), salvage_value (float), purchase_date (date),
funding_details_id (int). The dataset should have 1000 rows,
sorted by id. Each row should have the following
characteristics:
-- - id should be a unique integer and sequential starting at 1.
-- - name should be a random string of characters between 1 and
50 characters long.
-- - status should be a random selection from the following
valid asset statuses: in use, in storage, disposed of, in
repair, in transit, other.
-- - category should be a random selection from the following
valid categories: hardware, software, other.
-- - cost should be a random float between 0 and 100000.
-- - useful_life should be a random int between 1 and 10.
-- - salvage_value should be a random float greater than 0 but
less than the cost of the asset.
-- - purchase_date should be a random date between 1/1/2019 and

```

```
12/31/2022.  
-- - funding_details_id should be a random integer either 1 or  
2.  
-- The dataset should be saved as a CSV file named assets.csv in  
the data directory. The file should have a header row and the  
columns should have the following data types: id (int), name  
(varchar), status (varchar), category (varchar), cost (float),  
useful_life (float), salvage_value (float), funding_details_id  
(int)
```

The solution that Copilot provides is novel. It builds a large series using a Postgres built-in function, meaning that this would not be a portable solution. However, given that this is the database we will use, this is an appropriate enough solution. The resultant dataset is refined. We would have gotten better results if we had used Python and asked for Copilot's assistance in coding a script to generate a file to load into Postgres. However, given, that this dataset is only for playing out with the application, we do not need to be overly concerned with the data quality for now. Although, in the real world, data quality is everything.

Listing 5.9 Copilot's response: An insert statement built off of a series

```
insert into item.assets (id, name, status, category, cost,  
useful_life, salvage_value, purchase_date, funding_details_id)  
select  
    id,  
    name,  
    status,  
    category,  
    cost,  
    useful_life,  
    salvage_value,  
    purchase_date,  
    funding_details_id  
from (  
    select  
        row_number() over (order by random()) as id,  
        md5(random()::text) as name,  
        case  
            when random() < 0.2 then 'in use'  
            when random() < 0.4 then 'in storage'  
            when random() < 0.6 then 'disposed of'
```

```

        when random() < 0.8 then 'in repair'
        when random() < 0.9 then 'in transit'
        else 'other'
    end as status,
    case
        when random() < 0.5 then 'hardware'
        when random() < 0.9 then 'software'
        else 'other'
    end as category,
    random() * 100000 as cost,
    (random() * 100)::int as useful_life,
    random() * (random() * 100000) as salvage_value,
    -- generate a random date between 1/1/2019 and
12/31/2022
    -- this does not work please fix
    -- '2019-01-01'::date + random() * ('2022-12-31'::date -
'2019-01-01'::date) as purchase_date,
    '2019-01-01'::date + (random() * (DATE '2022-12-31' -
DATE '2019-01-01')::integer)::integer as purchase_date
    case
        when random() < 0.5 then 1
        else 2
    end as funding_details_id
    from generate_series(1, 1000)
) as assets;

```

If we switch back to ChatGPT for just minute, we can get a second opinion as to how to create such a dataset. ChatGPT suggests the Python library faker. The faker package is used to generate fake data, like common English first names. numpy is used to generate the random float values for cost, useful life, and salvage value. pandas is used to manage the data in a DataFrame (the table). Additionally, we could save the DataFrame to a CSV file, using the method `df.to_csv('assets.csv', index=False)`.

Listing 5.10 ChatGPT suggests Faker to generate the fake dataset

```

import pandas as pd
import numpy as np
import random
from faker import Faker

# Create Faker object

```

```

fake = Faker()

# Define our valid statuses, categories, funding_details and
depreciation_strategies
valid_statuses = ['in use', 'in storage', 'disposed', 'in
repair', 'in transit', 'other']
valid_categories = ['hardware']
valid_funding_details = ['purchased']
valid_depreciation_strategies = ['straight line']

# Generate the data
data = {
    'id': list(range(1, 1001)),
    'name': [fake.first_name() for _ in range(1000)],
    'status': [random.choice(valid_statuses) for _ in
range(1000)],
    'category': [random.choice(valid_categories) for _ in
range(1000)],
    'cost': np.random.uniform(0, 100000, 1000),
    'useful_life': np.random.uniform(1, 10, 1000),
    'salvage_value': np.random.uniform(0, 10000, 1000),
    'funding_details': [random.choice(valid_funding_details) for
_ in range(1000)],
    'depreciation_strategy':
[random.choice(valid_depreciation_strategies) for _ in
range(1000)],
}
}

# Create DataFrame
df = pd.DataFrame(data)

# Ensure the DataFrame is sorted by id
df = df.sort_values('id')

# Show the DataFrame
print(df)

```

For each of these assets, we will need funding details as well; how they were financed (purchased in this case), and the depreciation details.

Unsurprisingly, we get a similar solution from Copilot: generate a series of entries using a similar prompt as what we used for the assets. We will need to ensure that for each of the asset identifiers (1-1000), we have a corresponding funding details entry. Otherwise, we would risk getting null pointers when running out code.

Listing 5.11 Creating a dataset for the funding_details table

```
-- Generate a dataset of funding details for an ITAM system. The
dataset should include the following columns: id (int), name
(varchar), depreciation_strategy_id (int), depreciation_rate
(float). The dataset should have 1000 rows, sorted by id.
-- Each row should have the following characteristics:
-- - id should be a unique integer and sequential starting at 1.
-- - name should be a random string of characters between 1 and
10 characters long.
-- - depreciation_strategy_id should be a random integer either
1 or 2.
-- - depreciation_rate should be a random float between 0 and
.4.

insert into itam.funding_details (id, name,
depreciation_strategy_id, depreciation_rate)
select
    id,
    name,
    depreciation_strategy_id,
    depreciation_rate
from (
    select
        row_number() over (order by random()) as id,
        md5(random()::text) as name,
        case
            when random() < 0.5 then 1
            else 2
        end as depreciation_strategy_id,
        random() * 0.4 as depreciation_rate
    from generate_series(1, 1000)
) as funding_details;
```

With the dataset generated and stored in the database, we should be able to wire up the remainder of our application to store and display assets using the REST APIs. However, since we had previously stripped out all of the metadata for SQLAlchemy during our build phase (see previous chapter), we need a way to wire this metadata with our adapters differently.

With this, we have reached the edge of Copilot's capabilities. We are perplexed by what comes next; how we can solve our most recent dilemma. Tempting as it is, we cannot give up and go home. Therefore, it is time to

introduce the most recent addition to the Copilot product suite: Copilot Chat. Copilot Chat is an embedded GPT-4 model in your IDE (only currently supported by Visual Studio Code). We shall open the chat dialog and ask how to keep our business model clean but still use SQLAlchemy's ORM (Object Relational Model) features.

Figure 5.1 GitHub Copilot Chat's solution for how to solve our most recent quandary



If you don't want to include any SQLAlchemy properties in your domain model, you can create a separate data access layer (DAL) that handles all interactions with the database. This way, your domain model can remain independent of any specific database implementation.

Here's an example of how you can create a DAL using SQLAlchemy:

Copilot Chat suggests that we create a separate Data Access Layer. This approach maps nicely onto the Ports and Adapters approach we have used thus far. In addition, Copilot Chat recommended modeling these classes similarly to the domain classes but including the metadata required for ORM functionality to work correctly. The resultant code is in Listing 5.12.

Listing 5.12 ORM support outside of the domain classes.

```
Base = declarative_base()

class FundingDetailsModel(Base):
    __tablename__ = 'funding_details'
    id = Column(Integer, primary_key=True)
    depreciation_rate = Column(Float)
    depreciation_strategy_id = Column(Integer)

    def get_depreciation_strategy(self) -> DepreciationStrategy:
        if self.depreciation_strategy_id is 1:
            return StraightLineDepreciationStrategy()
        else:
            return DoubleDecliningDepreciationStrategy()
```

```

class AssetModel(Base):
    __tablename__ = 'assets'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    status = Column(String)
    category = Column(String)
    cost = Column(Float)
    useful_life = Column(Float)
    salvage_value = Column(Float)
    purchase_date = Column(Date)
    funding_details_id = Column(Integer,
        ForeignKey('funding_details.id'))
    funding_details = relationship('FundingDetailsModel')

```

Now that the external model classes have been created, we must map these ORM instances to our domain model before returning them to the system's core. While this might seem like over-engineered code for such a simple application, this gives us great flexibility in how our domain model can operate. For example, our model can perform complex operations beyond just CRUD. We would be limited to these operations if we kept our domain model identity to the model used in the Data Access Layer.

Next, we will use Copilot and Copilot Chat to explore incorporating event-driven ideas into our application. Event-driven concepts will allow us to track our IT assets in real-time: their location, status, and market value, for example.

5.2 Monitoring our assets in real time with Kafka

We will monitor our assets in real-time to motivate our exploration of using Generative AIs in conjunction with event-driven architecture. We shall take it as a given that some system external to the Information Security Asset Management system fires events as our assets move from one location to another.

To delve into ITAM events, we will need to configure a few additional services. In this case, we will use Apache Kafka. Apache Kafka is a distributed streaming platform that is used for building real-time data pipelines and streaming apps. It's designed to handle data streams from

multiple sources and deliver them to multiple consumers, effectively acting as a middleman for our real-time data.

To start we will ask Copilot Chat how to run Kafka locally, using Docker. Apache Kafka has an undeserved reputation for being difficult to install and configure. Running in Docker will allow us to side-step this controversy. Using Copilot Chat we can produce a docker compose file. However, as is often the case, the versions are very old to the point of not supporting some hardware. Listing 5.13 is an updated listing from Confluent's (the company that offers commercial support for Kafka) official GitHub repository. Notice that the docker-compose file's content includes both Kafka and Zookeeper. Zookeeper is a distributed coordination service that Kafka uses to manage and coordinate the brokers within the cluster, at least for now. Future versions aim to remove dependency on Zookeeper.

Listing 5.13 Docker-Compose file to launch Kafka with Zookeeper

```
version: '2.1'

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.3.2
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_SERVER_ID: 1
      ZOOKEEPER_SERVERS: zoo1:2888:3888

  kafka:
    image: confluentinc/cp-kafka:7.3.2
    hostname: kafka
    container_name: kafka
    ports:
      - "9092:9092"
      - "29092:29092"
      - "9999:9999"
    environment:
      KAFKA_ADVERTISED_LISTENERS:
        INTERNAL://kafka:19092, EXTERNAL://${DOCKER_HOST_IP:-127.0.0.1}:9
```

```

092, DOCKER://host.docker.internal:29092
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
INTERNAL:PLAINTEXT, EXTERNAL:PLAINTEXT, DOCKER:PLAINTEXT
    KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
    KAFKA_ZOOKEEPER_CONNECT: "zookeeper:2181"
    KAFKA_BROKER_ID: 1
    KAFKA_LOG4J_LOGGERS:
"kafka.controller=INFO, kafka.producer.async.DefaultEventHandler=
INFO, state.change.logger=INFO"
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
    KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
    KAFKA_JMX_PORT: 9999
    KAFKA_JMX_HOSTNAME: ${DOCKER_HOST_IP:-127.0.0.1}
    KAFKA_AUTHORIZER_CLASS_NAME:
kafka.security.authorizer.AclAuthorizer
    KAFKA_ALLOW_EVERYONE_IF_NO_ACL_FOUND: "true"
depends_on:
- zookeeper

```

With our new Kafka instance running, we now need a consumer that will pull the updated locations off the topic asset_location (which we share create shortly) and update the inventory in the database. Again, we can ask Copilot Chat to provide us with a suggestion as to how to do this.

Listing 5.14 Prompt to Copilot Chat asking how to use Python to subscribe to the topic

In our ITAM project, we would like a Python consumer for the assets_location Kafka topic. Once it receives a new updated location, it should update the domain model to reflect the new location. How should we do this?

The code that Copilot Chat generates creates a consumer, listens to the topic, and uses a reference to the AssetManager class to update the location of the affected asset.

Listing 5.15 Copilot Chat code to monitor assets' locations

```

from kafka import KafkaConsumer
from asset_manager import AssetManager

```

```

consumer = KafkaConsumer(
    'asset_location',
    bootstrap_servers=['localhost:9092'],
    auto_offset_reset='earliest',
    enable_auto_commit=True,
    group_id='my-group',
    value_deserializer=lambda x: tuple(map(float, x.decode('utf-
8').split(','))))
)

asset_manager = AssetManager()

for message in consumer:
    asset_id, latitude, longitude = message.value
    asset_manager.update_asset_location(asset_id, (latitude,
longitude))

```

However, should the adapter be aware of the AssetManager? While the AssetManager does not directly sit inside the domain model, it is a critical intermediary. This coupling level between the AssetManager and the adapter could arguably violate the hexagon in the hexagonal architecture model. Let's have Copilot Chat weigh in. We can ask it: "Given that this project uses hexagonal architecture, is it a good idea to have an adapter aware of the AssetManager?"

Figure 5.2 GitHub Copilot Chat's acceptance that it is, in fact, imperfect



GitHub Copilot

In a hexagonal architecture, the adapters are responsible for translating external input and output to and from the domain model. The domain model should not depend on the adapters, but the adapters may depend on the domain model.

In the case of a Kafka adapter that listens for messages on the `asset_location` topic and updates the domain model with the new location, it is reasonable for the adapter to depend on the `AssetManager` class. The adapter would translate the Kafka message into a domain event that the `AssetManager` can handle, and then call the appropriate method on the `AssetManager` to update the domain model.

However, it is important to ensure that the adapter does not tightly couple with the `AssetManager` class. One way to achieve this is to define an interface or protocol in the domain layer that the adapter can depend on, rather than depending directly on the `AssetManager` class. The interface or protocol would define the methods that the adapter can call to update the domain model, and the `AssetManager` class would implement this interface or protocol.

This approach allows the domain model to remain decoupled from the adapters, while still allowing the adapters to interact with the domain model in a controlled and well-defined way.

It would appear to be the case that Copilot Chat agrees that this would indeed be considered a breach of the proper contraction responsibility of an adapter. We could add a new port to our `AssetManager` class that could use the Kafka port. However, let's see if Copilot Chat has any other suggestions: “We do not want the Kafka consumer to interact directly with the `AssetManager`. Are there any ways to accomplish this?” Copilot Chat suggests that we apply the `Mediator` pattern, which resembles a port and adapter combination.

Mediator Pattern

The Mediator design pattern is a behavioral pattern that promotes loose coupling between objects by encapsulating their interactions within a mediator object. The mediator object acts as a central hub that facilitates communication between objects without the objects having to know about each other.

Let us first take a look at the Mediator class that Copilot Chat generated. The class would sit between the AssetManager and the Kafka consumer.

Listing 5.16 The Mediator class that will mediate the interaction between the AssetManager and the Kafka consumer

```
class AssetLocationMediator:  
    def __init__(self):  
        self.handlers = {}  
  
    def register_handler(self, event_type, handler):  
        if event_type not in self.handlers:  
            self.handlers[event_type] = []  
        self.handlers[event_type].append(handler)  
  
    def publish(self, event):  
        event_type = type(event)  
        if event_type in self.handlers:  
            for handler in self.handlers[event_type]:  
                handler(event)
```

This implementation closely resembles a message queue or message publisher, which is precisely the point: decoupling.

Note

Eagle-eyed readers would likely notice that we are playing a little fast and loose with the distinction between an Asset and an asset of type Hardware. In the original domain model, only Hardware had a location. Generally, one does not think of Software as having a location. Of course, you could say that the software is installed in its location, but it is arguably how convincing one finds this argument. However, as this project continues, the reason for the flattening of the domain model is for simplicity's sake, as polymorphic structures in a persistence layer is a distractingly complex topic.

Now that we have a strategy to decouple the Kafka consumer with the AssetManager, we should update the Kafka consumer to take advantage of it. We will need to pass the mediator into the class in its constructor. This way the AssetManager and the consumer will have access to the same

instance, and messages can freely flow back and forth; or rather, in this case, the flow will be unidirectional. You should note that we intend to read and write JSON on this topic, so we will need to have our value deserializer understand this.

Listing 5.17 Incorporating the mediator into the Kafka consumer class

```
from kafka import KafkaConsumer
from itam.domain.events.asset_location_updated import
AssetLocationUpdated
import json

class AssetLocationKafkaConsumer:
    def __init__(self, mediator):
        self.mediator = mediator

        self.consumer = KafkaConsumer(
            'asset_location',
            bootstrap_servers=['localhost:9092'],
            enable_auto_commit=True,
            group_id='itam-group',
            value_deserializer=lambda m:
            json.loads(m.decode('utf-8'))
        )

    def poll(self):
        print("Polling for asset location updates")
        for message in self.consumer:
            asset_id = message.value['asset_id']
            latitude = message.value['latitude']
            longitude = message.value['longitude']
            timestamp = message.value['timestamp']
            event = AssetLocationUpdated(asset_id, latitude,
            longitude, timestamp)
            self.mediator.publish(event)
```

Next, we shall examine the changes that the `AssetManager` class require to incorporate the ability to track these locations. You should note that to get this project to run in its entirely, you would need to modify the `AssetManager`, `SQLAlchemyAssetRepository`, and `Asset` classes, as well as create a new table in your database called `itam.asset_locations`. The

complete and updated source code is available in the GitHub repository for this book. For now, we shall just focus on the changes needed to get the events flowing through our system and use the repository for reference if the reader so chooses.

Figure 5.3 AssetManager requires the additional of another constructor parameter and a method to handle the updates to its locations objects

AssetManager
-BaseRepository[Asset] _repository
-AssetLocationMediator mediator
+__init__(base_repository: BaseRepository[Asset], mediator: AssetLocationMediator): None
+update_asset_location(event: AssetLocationUpdated): None

There are two required changes for the AssetManager class: First, we need to add the AssetLocationMediator added to the constructor, registering it to handle the AssetLocationUpdated event. And secondly, we need to add a method that will handle this event. In this case, we call the method update_asset_location. The abridged code is below.

Listing 5.18 The updated constructor and an event handler for the AssetManager class

```
from item.infrastructure.mediators.asset_location_mediator  
import  
  
class AssetManager:  
    def __init__(self, base_repository: BaseRepository[Asset],
```

```

mediator: AssetLocationMediator):
    self._repository = base_repository
    self.mediator = mediator
    self.mediator.register_handler(AssetLocationUpdated,
self.update_asset_location)

    def update_asset_location(self, event: AssetLocationUpdated)
-> None:
        asset = self.read(event.asset_id)
        asset.add_location(event.latitude, event.longitude,
event.timestamp)
        #self.update(asset)
        print(f"Asset {asset.id} location updated to
{event.latitude}, {event.longitude} at {event.timestamp}")

```

The `add_location` method of the `Asset` class merely appends the new Location to the end of a list of Locations. More sophisticated domain models might include a `current_location` attribute, relegating the rest to a list of historical locations; however, given that we are trying to get our events flowing through the system, it behooves us to keep things simple.

There is only one final item on our todo list: create the topic. How does one create a topic? That is a good question. Thankfully, all the tools we need are available in our running Docker container. So, let's log into our Kafka Docker instance. We use the following command (assuming that your Docker instance is named `kafka`): `docker exec -it kafka /bin/bash`.

The first thing that should check is if any topics are already created. We can do that with the following command: `kafka-topics --list --bootstrap-server localhost:9092`. This command would list all of the existing topics, running on this Kafka cluster. As you can see, there aren't any.

Given the need for a topic, let's create it. You would use the following command: `kafka-topics --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic asset_location`. If you re-run the `kafka-topics --list` command again, you will see the new topic. The partitions and replication-factor instructions we included in the create topic command inform Kafka that we want one partition and a replication factor of one. If we were setting this up

for production, or any purpose other than testing, we would likely want them to be greater than that to ensure availability of data.

Table 5.1 Summary of Kafka console commands

Action	Command
Create	kafka-topics --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic asset_location
Read	kafka-console-consumer --broker-list localhost:9092 --topic asset_location -from-beginning
Write	kafka-console-producer --broker-list localhost:9092 --topic asset_location
Delete	kafka-topics --delete --topic asset_location --bootstrap-server localhost:9092
List	kafka-topics --list --bootstrap-server localhost:9092

Now comes the fun part, observing the application in action. Kafka comes with a console produce that will allow us to publish messages to Kafka from standard in. To do this, we should launch the console producer with the follow command: `kafka-console-producer --broker-list localhost:9092 --topic asset_location`. You will enter an interactive session allowing you to publish a message with every line. Let's publish a few messages simulating our asset moving around or near Chicago.

Listing 5.19 Entries for the Kafka console producer

```
{"asset_id": 1, "latitude": 41.8781, "longitude": -87.6298,  
"timestamp": "2022-01-01T00:00:00Z"}  
{"asset_id": 1, "latitude": 41.9000, "longitude": -87.6244,  
"timestamp": "2022-01-01T00:10:00Z"}  
{"asset_id": 1, "latitude": 41.8676, "longitude": -87.6270,  
"timestamp": "2022-01-01T00:20:00Z"}  
{"asset_id": 1, "latitude": 41.8788, "longitude": -87.6359,  
"timestamp": "2022-01-01T00:30:00Z"}  
{"asset_id": 1, "latitude": 41.8740, "longitude": -87.6298,  
"timestamp": "2022-01-01T00:40:00Z"}  
{"asset_id": 1, "latitude": 41.8800, "longitude": -87.6300,  
"timestamp": "2022-01-01T00:50:00Z"}  
{"asset_id": 1, "latitude": 41.8775, "longitude": -87.6244,  
"timestamp": "2022-01-01T01:00:00Z"}  
{"asset_id": 1, "latitude": 41.8745, "longitude": -87.6328,  
"timestamp": "2022-01-01T01:10:00Z"}  
{"asset_id": 1, "latitude": 41.8825, "longitude": -87.6298,  
"timestamp": "2022-01-01T01:20:00Z"}  
{"asset_id": 1, "latitude": 41.8800, "longitude": -87.6250,  
"timestamp": "2022-01-01T01:30:00Z"}
```

As you enter these messages, you should see the output from your application indicating that the location has been updated.

For completeness's sake, there is one more command that you should be aware of: you might make a mistake when entering these messages. An invalid message could potentially break your consumer. One possible solution is to delete the topic. Deleting a topic might sound dramatic, but it will solve the issue. So here is that command: `kafka-topics --delete --topic asset_location --bootstrap-server localhost:9092`.

In this section, we have added the ability to see changes in location of our Assets in real-time tracking using Apache Kafka. In the final section of this chapter, we will work with Copilot Chat to extend the capacity by monitoring our Assets in real-time, attempting to determine if they are where they should be. Again, we will explore using Spark and Kafka together to accomplish this analysis. Once completed, we will win the thanks for our InfoSec team who fear that too much of our core business and intellectual property exists on and within these Assets.

5.3 Analyzing, Learning, and Tracking with Apache Spark

Real-time tracking of assets is a business-critical function. Your IT assets contain sensitive business data, client lists, sales figures, Profits and Loss (PnL) projections, and sales strategies, amongst many other items. A lost asset can be an existential event for a company. Therefore, careful management and monitoring is priority one for many InfoSec professionals. In this section, we aim to make their jobs substantially easier. Modern data platforms make it trivial to track your assets in real time and send notifications should questionable conditions arise. Let's get into it.

Apache Spark is a powerful, open-source data processing engine built around speed, ease of use, and sophisticated analytics. It was developed to provide an improved alternative to MapReduce for processing big data sets and can handle batch and real-time analytics. Spark provides APIs for Scala, Java, Python, and R and a built-in module for SQL queries. Its core data structure, the Resilient Distributed Dataset (RDD), enables fault-tolerant operation and allows data to be processed in parallel across a cluster of computers.

Spark also includes several libraries to broaden its capabilities, including MLlib for machine learning, Spark Streaming for processing live data streams, and Spark SQL and DataFrames for processing structured data. These tools make it well-suited for tasks ranging from machine learning to real-time data streaming and batch processing. Its in-memory processing capabilities make Spark significantly faster than its predecessor, making it a popular choice for big data processing.

First, we will ask Copilot Chat to recommend a strategy for using Apache Spark to track our assets.

Listing 5.20 The prompt to ask Copilot Chat how best to track out Assets in real-time

Let's imagine that our company is in Chicago. We want to create a class called AssetLocationSparkAdapter that will continuously stream AssetLocation json messages from a Kafka topic called asset_location. An AssetLocation has the following attributes asset_id: int, latitude: float, longitude: float, and timestamp: datetime. The AssetLocations are stored in a Postgres database in a table called item.asset_locations.

AssetLocationSparkAdapter should calculate if the new AssetLocation is more than 25 miles from Chicago. If it is, then it should write a message to the console. This should be using the latest version of Spark. How would we do this?

Copilot Chat generates the class that you should be able to put into a file called asset_location_spark_adapter.py in the infrastructure package. Helpfully, it also includes comments for each line, so you should find the generated code easy to follow. The import statements include the Spark libraries, as well as the geopy.

Listing 5.21 The imports required to run Spark

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import from_json, col, expr
from pyspark.sql.types import StructType, StructField,
IntegerType, DoubleType, TimestampType
from geopy.distance import distance
```

The class begins with an overstuffed constructor that defines the schema Spark will use when it translates the JSON to a DataFrame.

Note

The AssetLocationSparkAdapter, as defined, is a blocking process. Therefore, your FastAPI application will not “fully” boot until the Spark process has been killed. You would want this to be a standalone process, or you would need to introduce an asynchronous framework to have these two processes run concomitantly.

Next, it will start up a local Spark instance/session that will allow Spark to connect to the Kafka topic and continuously stream in the records.

Listing 5.22 The AssessLocationSparkAdapter, which is responsible for processing the Kafka topic and generating notifications

```
class AssetLocationSparkAdapter:  
    def __init__(self):  
        # Define the schema for the incoming JSON data  
        self.schema = StructType([  
            StructField("asset_id", IntegerType()),  
            StructField("latitude", DoubleType()),  
            StructField("longitude", DoubleType()),  
            StructField("timestamp", TimestampType())  
        ])  
  
        # Create a SparkSession  
        self.spark = SparkSession.builder \  
            .appName("AssetLocationSparkAdapter") \  
            .getOrCreate()  
  
        # Create a streaming DataFrame from the asset_location  
topic  
        self.df = self.spark \  
            .readStream \  
            .format("kafka") \  
            .option("kafka.bootstrap.servers", "localhost:9092")  
        \  
            .option("subscribe", "asset_location") \  
            .option("startingOffsets", "earliest") \  
            .load() \  
            .selectExpr("CAST(value AS STRING)")  
  
        # Parse the incoming JSON data  
        self.parsed_stream = self.df \  
            .select(from_json(col("value"),  
self.schema).alias("data")) \  
            .select("data.*")
```

The final section of the AssetLocationSparkAdapter class will calculate the distance from the asset's current location to Chicago's. If the difference is greater than 25 miles, then it will send the result set to the console. Additionally, it provides a method to start and stop the adapter.

Listing 5.23 The AssessLocationSparkAdapter calculating the distance from the current Asset's location from Chicago

```

        # Calculate the distance between the current location
        and Chicago for each asset
        self.distance = self.parsed_stream \
            .withColumn("distance",
            expr("calculate_distance(latitude, longitude, 41.8781,
            -87.6298)")) \
            .select(col("asset_id"), col("timestamp"),
            col("distance")) \
            .filter(col("distance") > 25)

        # Write the results to the console
        self.query = self.distance \
            .writeStream \
            .outputMode("append") \
            .format("console") \
            .start()

    def run(self):
        # Start the streaming query
        self.query.awaitTermination()

    def stop(self):
        # Stop the streaming query and SparkSession
        self.query.stop()
        self.spark.stop()

```

The calculate_distance method takes the logitude and latitude of the asset's location and determines the distance from Chicaogo using the geopy.distance function.

Listing 5.24 The function that Spark uses to calculate the distance between Chi-town and your Asset

```

def calculate_distance(lat1, lon1, lat2, lon2):
    return distance((lat1, lon1), (lat2, lon2)).miles

```

In this instance, the code that Copilot Chat produced had some issues preventing it from running locally. After running it locally, encountering these issues, and trolling Stack Overflow, you would have found a solution to the two main issues with the code: a missing environmental variable for running locally, and failing to register your UDF (User Defined Function).

Thankfully, you do not need to do the testing and research, as a solution is provided in listing 5.23.

Listing 5.25 Edits required to run the application locally

```
os.environ['PYSPARK_SUBMIT_ARGS'] = '--packages  
org.apache.spark:spark-streaming-kafka-0-  
10_2.12:3.2.0,org.apache.spark:spark-sql-kafka-0-10_2.12:3.2.0  
pyspark-shell'  
  
class AssetLocationSparkAdapter:  
    def __init__(self):  
        # Create a SparkSession  
        self.spark = SparkSession.builder \  
            .appName("AssetLocationSparkAdapter") \  
            .getOrCreate()  
        self.spark.udf.register("calculate_distance",  
        calculate_distance)
```

Finally, to run your Spark application, you would update `main.py` with the following code in the `main` function.

Listing 5.26 Updates to the `main` function

```
if __name__ == "__main__":  
    adapter = AssetLocationSparkAdapter()  
    adapter.run()
```

As you enter locations for your asset into the Kafka console producer that are further than twenty-five miles from downtown Chicago, you will notice entries get written to the console. It would be trivial to update the class to output these results to Twilio's SMS API or an email service such as SendGrid.

Listing 5.27 The streaming output from your asset location

```
+-----+-----+-----+  
|asset_id|      timestamp|      distance|  
+-----+-----+-----+
```

1	2021-12-31 20:30:00	712.8314662207446
---	---------------------	-------------------

Congratulations, you are tracking your assets in real-time and sending real-time alerts should your corporate resources grow legs and walk away.

5.4 Summary

- GitHub Copilot Chat is an innovative tool that brings together the comprehensive language understanding of ChatGPT and the handy features of Copilot. It's a noteworthy development in the realm of programming assistance, particularly for providing detailed and contextually relevant suggestions in real-time, fostering a more efficient coding experience.
- The Mediator design pattern is a distinct behavioral pattern that facilitates a high level of decoupling between objects, thus enhancing the modularity of your code. By encompassing the interactions between objects within a mediator object, objects can communicate indirectly, which reduces dependencies and promotes code reusability and ease of modification.
- Apache Kafka serves as a robust, distributed streaming platform engineered for creating real-time data pipelines and streaming applications. It can effectively handle data streams from a multitude of sources and transmit them to various consumers, making it an ideal solution for use cases that require handling substantial volumes of real-time or near-real-time data. It's important to remember that Kafka is optimized for append-only, immutable data and not for use cases that need record updates or deletions, or complex querying.
- Apache Spark stands as a high-performance, distributed data processing engine renowned for its speed, user-friendliness, and advanced analytics capabilities. It's highly suitable for scenarios necessitating real-time data processing or for operations on enormous datasets. However, for simpler tasks such as basic analytics or straightforward aggregations, a traditional relational database could be a more appropriate choice.

- Generative AI, despite its rapid evolution, is not infallible. It's crucial to meticulously review all generated output to ensure it aligns with your specific requirements and quality standards. While generative AI is not a substitute for deep domain knowledge or coding expertise, it significantly enhances productivity by providing valuable insights and reducing the time spent on routine tasks.

6 Testing, Assessing, and Explaining with Large Language Models

This chapter covers

- Drafting unit tests with ease
- Generating integration tests
- Determining code quality and coverage
- Assessing software complexity
- Translating code and text

This chapter will explore a critical aspect of software engineering: testing. The act of testing software serves multiple essential purposes. First and foremost, it aids in the identification of bugs, errors, and issues that could potentially impact the software's functionality, usability, or performance. Furthermore, it ensures that the software adheres to the required quality standards. By conducting thorough tests, we can verify if the software meets the specified requirements, functions as intended, and produces the expected outcomes. Through comprehensive testing, developers can evaluate the software's reliability, accuracy, efficiency, security, and compatibility across various platforms and environments. Detecting and resolving software defects early in the development process can result in significant time and cost savings.

Once we have finished formulating our tests, we will evaluate our code's quality. You will be introduced to several metrics that prove helpful in assessing software quality and complexity. Additionally, if we need clarification on the purpose of our code or are reviewing it for the first time, we will seek an explanation to ensure a thorough understanding.

6.1 Testing, Testing...One, Two, Three Types

Testing plays a vital role in software engineering; therefore, we will explore various types of testing in detail. This includes unit tests, integration tests,

and behavior tests. To start, we will leverage Copilot Chat to assist us in creating a *unit test*.

Unit Testing

A unit test is a type of testing that focuses on testing individual components or units of code to ensure they function correctly in isolation. Developers usually perform it and help identify bugs and issues within specific software units.

6.1.1 Unit Testing

In this section, we will create unit tests to test our software components. There are several unit testing frameworks available for Python. Each has its unique features and is suitable for different scenarios. We will examine each of them briefly before settling on a specific framework based off of the recommendation provided by our AI tool.

The first framework is `unittest`: This is Python's standard library for creating unit tests. It comes bundled with Python and doesn't need to be installed separately. `unittest` provides a rich set of assertions and is great for writing simple to complex test cases, but it can be quite verbose. `unittest` is a good choice for writing basic unit tests, especially if you don't want to introduce additional dependencies in your project. It's useful in any scenario where you need to confirm the functionality of individual units of code in isolation from the rest of the system.

Next, let's examine `pytest`: `pytest` is a popular third-party library that can be used for unit testing, although it's versatile enough to handle more than just unit tests. It requires less boilerplate code than `unittest` and has powerful features like fixtures for setup and teardown, parameterized testing, and the ability to run `unittest` and `nose` test suites. `pytest` is great for both simple and complex unit test cases. It's also useful for functional and integration tests. If you value simplicity and ease of use, and your project is not restricted to using only the Python standard library, `pytest` is an excellent choice.

Next, we have nose2, the successor to the deprecated "nose" testing framework. It extends unittest and makes testing easier. It's known for its test discovery feature, which automatically finds your project's tests so you don't have to manually list them. nose2 is good for larger projects where test discovery can save time. Like pytest, it can run unittest test suites, so it's also a good choice if you're migrating from unittest but want to keep your existing tests.

Finally, there is doctest. This is another module that's part of the Python standard library. It's not a full-featured testing framework like the others, but it allows you to write tests directly in your docstrings. doctest is best suited for simple cases where you want to demonstrate how to use a function or module and confirm that the example code works as expected. It's a great way to ensure your documentation stays up to date with your code.

Copilot Chat recommended that we use unittest, so we shall. You might try to ask it yourself and see if it makes a different conclusion, which it might, depending on how you frame the question.

First, we will define test cases by creating classes that inherit from the unittest.TestCase class. In writing our first unit test we will uncover a rather pernicious bug that has thus far eluded us. We shall get to that shortly. Finally, we are going to test the correctness of the DepartmentStatisticsVisitor class.

We start with a prompt asking Copilot Chat to create a unit test for the DepartmentStatisticsVisitor. The response will be a reminder of the importance of context. Copilot Chat, given that we began a new chat session, will attempt to create this unit test, but it lacks the definition of any of our classes.

Listing 6.1 The first attempt at defining a unit test for DepartmentStatisticsVisitor

```
import unittest
from department_visitor import DepartmentStatisticsVisitor
```

```

class TestDepartmentStatisticsVisitor(unittest.TestCase):
    def test_visit_department(self):
        visitor = DepartmentStatisticsVisitor()
        department = {'name': 'Sales', 'employees': [{'name': 'John', 'salary': 50000}, {'name': 'Jane', 'salary': 60000}]}
        visitor.visit_department(department)
        self.assertEqual(visitor.total_salaries, 110000)
        self.assertEqual(visitor.num_employees, 2)
        self.assertEqual(visitor.avg_salary, 55000)

```

We should proceed by inputting definitions into Copilot Chat so that it can refine the test case. We can tell it more of the visitor class, as shown in the following snippet.

The `DepartmentStatisticsVisitor` class visits to determine the total asset depreciation based off of the asset allocation for each department

With this updated information, Copilot Chat is about to better estimate how our unit test should function.

Listing 6.1 The first attempt at defining a unit test for `DepartmentStatisticsVisitor`

```

import unittest
from department_visitor import DepartmentStatisticsVisitor,
Asset

class TestDepartmentStatisticsVisitor(unittest.TestCase):
    def test_visit(self):
        visitor = DepartmentStatisticsVisitor()
        asset1 = Asset(1000, 500, 0.5)
        asset2 = Asset(2000, 1000, 0.3)
        visitor.visit(asset1)
        visitor.visit(asset2)
        self.assertEqual(visitor.get_total_cost(), 3000)
        self.assertEqual(visitor.get_total_depreciation(), 1500)
        self.assertEqual(visitor.get_total_allocation(), 0.8)

```

The unit test is close to what is needed; however, the definition is `Asset` is incorrect. We should enter the proper definition from our `asset.py` class. Copilot Chat will continue to refine the unit test.

**Listing 6.2 The next attempt at defining a unit test for
DepartmentStatisticsVisitor now that Copilot Chat is aware of how to define
Asset**

```
import unittest
from datetime import date
from typing import List
from dataclasses import dataclass

@dataclass
class Location:
    name: str
    address: str

@dataclass
class Asset:
    id: int
    name: str
    status: str
    category: str
    cost: float
    useful_life: int
    salvage_value: float
    purchase_date: date
    locations: List[Location]

class TestAsset(unittest.TestCase):
    def test_asset(self):
        location1 = Location('Office', '123 Main St')
        location2 = Location('Warehouse', '456 Elm St')
        asset = Asset(1, 'Computer', 'In use', 'Hardware',
1000.0, 3, 100.0, date.today(), [location1, location2])
        self.assertEqual(asset.id, 1)
        self.assertEqual(asset.name, 'Computer')
        self.assertEqual(asset.status, 'In use')
        self.assertEqual(asset.category, 'Hardware')
        self.assertEqual(asset.cost, 1000.0)
        self.assertEqual(asset.useful_life, 3)
        self.assertEqual(asset.salvage_value, 100.0)
        self.assertEqual(asset.purchase_date, date.today())
        self.assertEqual(asset.locations, [location1,
location2])
```

This is closer, we should also remind Copilot Chat that the Asset class also has a class called `FundingDetails`. Once we provide it with the definition of `FundingDetails` and inform Copilot Chat that it should be part of Asset, we get closer.

Listing 6.3 The third attempt at a unit test, which comes reasonably close

```
import unittest
from department_visitor import DepartmentStatisticsVisitor,
Asset, FundingDetails
from datetime import date
from typing import List, Dict
from dataclasses import dataclass

@dataclass
class Location:
    name: str
    address: str

class TestDepartmentStatisticsVisitor(unittest.TestCase):
    def test_visit(self):
        visitor = DepartmentStatisticsVisitor()
        location1 = Location('Office', '123 Main St')
        location2 = Location('Warehouse', '456 Elm St')
        asset1 = Asset(1, 'Computer', 'In use', 'Hardware',
1000.0, 3, 100.0, date.today(), [location1, location2])
        asset2 = Asset(2, 'Printer', 'In use', 'Hardware',
500.0, 2, 50.0, date.today(), [location1])
        funding_details1 = FundingDetails(asset1, 'Straight
Line', 0.33, { 'IT': 0.5, 'HR': 0.5 })
        funding_details2 = FundingDetails(asset2, 'Double
Declining Balance', 0.25, { 'IT': 0.3, 'HR': 0.7 })
        visitor.visit(asset1, funding_details1)
        visitor.visit(asset2, funding_details2)
        self.assertEqual(visitor.get_total_cost(), 1500.0)
        self.assertEqual(visitor.get_total_depreciation(),
550.0)
        self.assertEqual(visitor.get_total_allocation(), 0.8)
```

All that remains is to manually update the import statements.

NOTE

Let this exercise be a cautionary tale in the usage of Large Language Models; coercing them to behave with respects to highly focused tasks can be very challenging to the point of not being worth the effort. In the real-world, no one would blame you for abandoning the prompt-engineering and go back to just coding out this test. However, with some persistence you could be able to build out a template library that you could use to build a suite of unit tests for similarly shaped classes. One additional note is that Copilot Chat can and will generate a test for the file in the editor window if you instruct it to “Generate (a) unit test for my code,” however, it will mock all objects/properties that are not directly part of the class under test. Depending on what you are attempting to test, the utility of this feature might be questionable.

What we discover, when we attempt to run this test, is that there is a *circular dependency* between visitor, asset, funding details, and depreciation strategy. A circular dependency is a situation in which two or more modules or components depend on each other directly or indirectly. In our case, when Python tries to instantiate the Asset, it tries to load the definition of FundingDetails.

We fix this by moving away from a direct instantiation or reference to the FundingDetails class.

Listing 6.4 The updated Asset class that does not directly reference the FundingDetails class

```
@dataclass
class Asset():
    id: int
    name: str
    status: str
    category: str
    cost: float
    useful_life: int
    salvage_value: float
    purchase_date: date
    locations: List[Location]
    funding_details: None or
    'itam.domain.funding_details.FundingDetails'
```

We will need to do the same to the `FundingDetails` class. It should not directly reference the `DepreciationStrategy` class.

Listing 6.5 The updated `FundingDetails` class that does not directly reference the `DepreciationStrategy` class

```
@dataclass
class FundingDetails:
    depreciation_rate: float
    department_allocations: Dict[Department, float]
    depreciation_strategy: DepreciationStrategy or
    'itam.domain.depreciation_strategy.DepreciationStrategy'
    asset: None or 'itam.domain.asset.Asset'
```

As we saw, we were able to create a unit test using Copilot Chat. However, we would likely have been able to create it with more ease had we written it without Copilot. The tool is surprisingly good at providing you guidance as to when and how to test your code, but the implantation (at least currently) leaves something to be desired.

In the real world, we would continue to add unit tests to build up a substantial body of tests. How many tests is substantial, you ask? We will explore this shortly. However, we should first turn our attention to the next type of test: the *integration test*.

Integration Testing

Integration testing involves testing the interaction between different components or modules of the software to ensure they work together seamlessly. It verifies that the integrated system functions as expected and detects any inconsistencies or communication problems between modules.

6.1.2 Integration Testing

In this section, we will develop an integration test that will allow us to test the end-to-end system. Thankfully, `fastapi` comes with its own test client, which will aid us in the creation of this test.

We begin by copying in the definition of our AssetController into the Copilot Chat window. We can then ask Copilot Chat how to create a integration test for this controller. Given that we included the routes in the definition, Copilot Chat should be able to provide us with a accurate integration tests. We will need to specify that we will use the InMemoryAssetRepository class or fix this after the test has been generated.

Listing 6.6 The Copilot Chat generated integration test of the AssetController

```
from fastapi.testclient import TestClient
from itam.infrastructure.adapters.memory_asset_repository import
MemoryAssetRepository
from itam.infrastructure.ports.asset_rest_port import
AssetRestPort
from itam.interfaces.asset_controller import AssetController

def test_get_assets():
    # Create an instance of the InMemoryAssetRepository
    asset_repository = InMemoryAssetRepository()

    # Add some assets to the repository
    asset_repository.create(Asset(name="Asset 1",
unit_cost=1000))
    asset_repository.create(Asset(name="Asset 2",
unit_cost=2000))

    # Create an instance of the AssetRestPort using the
MemoryAssetRepository
    asset_rest_port = AssetRestPort(asset_repository)

    # Create an instance of the AssetController using the
AssetRestPort
    asset_controller = AssetController(asset_rest_port)

    # Create a TestClient instance using the AssetController's
router
    client = TestClient(asset_controller.get_router())

    # Send a GET request to the /assets endpoint
    response = client.get("/assets")

    # Check that the response status code is 200 OK
    assert response.status_code == 200
```

```
# Check that the response body contains the expected assets
assert response.json() == [
    {"id": 1, "name": "Asset 1", "unit_cost": 1000},
    {"id": 2, "name": "Asset 2", "unit_cost": 2000},
```

We will now turn our attention to the final type of testing that we examine: *behavior testing*.

Behavior testing

Behavior testing is a type of testing that focuses on the behavior of a system as a whole, from the perspective of an end user. Behavior testing is typically used to test the functionality of a system and to ensure that it meets the requirements and specifications that have been defined for it.

6.1.3 Behavior Testing

All testing, in some sense is behavior testing, as tests verify the behavior of the system. However, behavior testing is unique in some respects. Let's summarize the different types of testing that we have encountered thus far and contrast this against behavior testes.

Unit testing is a type of testing that focuses on testing individual units or components of a system in isolation, typically using automated tests. Unit tests are designed to test the functionality of individual functions or methods, and to ensure that they behave correctly under a variety of conditions.

Integration testing, on the other hand, is a type of testing that focuses on testing the interactions between different components or units of a system. Integration testing is typically used to test the interfaces between different components or units, and to ensure that they work together correctly.

Integration testing can be performed manually or using automated tests, and it typically involves testing the interactions between different components or units of a system, rather than the system as a whole.

Behavioral testing focus is on defining the behavior of the software in terms of user stories or scenarios. These scenarios are written in a specific format called "Given-When-Then" (GWT) and are used to drive the development process. The GWT format describes the preconditions (Given), the actions (When), and the expected outcomes (Then) of a particular scenario.

As we progress with our testing, we may find that some behavior or components are difficult to set up in our test. Additionally, we may find it tricky to isolate the behavior of a particular object or module, and to test the interactions between different objects. To address this limitation, we can use a *mock object*.

Mock Objects

A mock object is a test double that simulates the behavior of a real object in a controlled way. They can also be used to simulate error conditions or edge cases that are difficult to reproduce with real objects. Mock objects can be created manually using a mocking library such as `unittest.mock` or `pytest-mock`. These libraries provide functions and classes for creating and configuring mock objects. Mock objects can be configured to return specific values or raise specific exceptions when their methods are called. They can also be used to record the calls made to their methods, so that you can verify that the correct methods were called with the correct parameters.

We will incorporate mock objects into the prompt to create a behavior test for the `AssetManager`.

Listing 6.7 The prompt to set up a behavior test for the AssetManager

We have a class called `AssetManager`. `AssetManager`'s constructor takes two class `BaseRepository` and `AssetLocationMediator`. The `BaseRepository` is a repository in which Assets are stored. We want to use `InMemoryAssetRepository`, which is a subclass of `BaseRepository` that stores the Assets in memory. Use a mock object for `AssetLocationMediator`. The `AssetManager` stores Assets using the method `create` and gets all of the Assets using the method `get_assets`. An asset has 10 required positional

arguments: 'id', 'name', 'status', 'category', 'cost', 'useful_life', 'salvage_value', 'purchase_date', 'locations', and 'funding_details'. Use named parameters to make it clear which param is going where. Create a BDD script that will add two Assets to the AssetManager, confirming that all Assets cost \$4000.00. This should be implemented in Python. Take it step by step, telling us the tools and libraries you will use.

Copilot Chat will tell us that we need to create a new file (called a feature file) called `asset_manager.feature`. In this file, we will define the AssetManager's attributes (and any supporting classes). For example, in the feature file defines a scenario that adds two assets to the AssetManager, confirms that the total cost of all assets is \$4000.00, and mocks the AssetLocationMediator class.

Listing 6.8 The definition of the `asset_manager.feature` file according to Copilot Chat

```
Feature: Asset Manager
  As an IT Asset Manager
  I want to be able to manage assets
  So that I can keep track of all IT assets in my organization

  Scenario: Add two assets to the Asset Manager
    Given the Asset Manager is running
    And the InMemoryAssetRepository is initialized
    And the AssetLocationMediator is mocked
    When I create an asset with a cost of $2000.00
    And I create another asset with a cost of $2000.00
    Then the total cost of all assets should be $4000.00
```

We should create a directory in the root of our project called `features`. We should place this file in that directory. Additionally, we should add a subdirectory called `steps`. We will get the steps in just a moment.

Next, we will create a new Python file called `steps.py` and place it in the `steps` directory. This Python file defines the step implementations for the scenario in the feature file. The AssetManager, InMemoryAssetRepository, AssetLocationMediator, and Asset classes are imported, and six functions are defined to correspond to the steps in the

scenario. These functions use the context object to store and retrieve data between steps. The `Mock` class from the `unittest.mock` library is used to create a mock object for the `AssetLocationMediator` class. The `Asset` class is instantiated using named parameters to clarify which parameter belong to which attribute. If we ask Copilot Chat for the contents of the `steps.py` file, we will see output similar to the following listing.

Listing 6.9 The implementation details for our behavior test as defined in `steps.py`

```
from behave import given, when, then
from item.service.asset_manager import AssetManager
from item.infrastructure.repository.in_memory_asset_repository
import InMemoryAssetRepository
from item.domain.asset import Asset
from item.infrastructure.mediators.asset_location_mediator
import AssetLocationMediator
from unittest.mock import Mock

@given('the Asset Manager is running')
def step_impl(context):
    context.asset_repository = InMemoryAssetRepository()
    context.asset_location_mediator =
    Mock(spec=AssetLocationMediator)
    context.asset_manager =
    AssetManager(context.asset_repository,
    context.asset_location_mediator)

@given('the InMemoryAssetRepository is initialized')
def step_impl(context):
    pass

@given('the AssetLocationMediator is mocked')
def step_impl(context):
    pass

@when('I create an asset with a cost of ${cost}')
def step_impl(context, cost):
    asset = Asset(id=1, name='Test Asset 1', status='Available',
category='Test Category', cost=float(cost), useful_life=5,
salvage_value=0, purchase_date='2022-01-01', locations=['Test
Location'], funding_details={'Test Funding': 1000})
    context.asset_manager.create(asset)
```

```

@when('I create another asset with a cost of ${cost}')
def step_impl(context, cost):
    asset = Asset(id=2, name='Test Asset 2', status='Available',
category='Test Category', cost=float(cost), useful_life=5,
salvage_value=0, purchase_date='2022-01-01', locations=['Test
Location'], funding_details={'Test Funding': 1000})
    context.asset_manager.create(asset)

@then('the total cost of all assets should be ${total_cost}')
def step_impl(context, total_cost):
    assets = context.asset_manager.get_assets()
    assert sum(asset.cost for asset in assets) ==
float(total_cost)

```

If you did not have it installed already, you should Install the behave library using pip: pip install behave. Additionally, you should add it to the requirements.txt file to ensure that it will get pulled in when we later build a deployable version of this application. We will run the behavior test by issuing the following command from the root directory of our project.

Listing 6.10 Running the behavior test with the resultant output

```

% behave features
Feature: Asset Manager # features/asset_manager.feature:1
  As an IT Asset Manager
    I want to be able to manage assets
    So that I can keep track of all IT assets in my organization
      Scenario: Add two assets to the Asset Manager          #
features/asset_manager.feature:6
  Given the Asset Manager is running                      #
features/steps/steps.py:8 0.000s
  And the InMemoryAssetRepository is initialized        #
features/steps/steps.py:14 0.000s
  And the AssetLocationMediator is mocked              #
features/steps/steps.py:18 0.000s
  When I create an asset with a cost of $2000.00      #
features/steps/steps.py:22 0.000s
  And I create another asset with a cost of $2000.00   #
features/steps/steps.py:27 0.000s
  Then the total cost of all assets should be $4000.00 #
features/steps/steps.py:32 0.000s

1 feature passed, 0 failed, 0 skipped

```

```
1 scenario passed, 0 failed, 0 skipped
6 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.001s
```

In this section, we have laid a foundation for good software development by using three types of tests: unit, integration, and behavior. Now, one may quibble that it came very late in the development lifecycle of this project. One would not be wrong. In the real world, we would develop our tests as we develop our code. Some might argue that we should build our test before our code. You may or may not hold this belief, but either way you should test early, and test often.

In the next section of the book, we will dive into some metrics that can be used to determine the overall quality of our software, and we will ask Copilot to help us assess the quality of our code thus far.

6.2 Assessing Quality

Understanding the performance, reliability, maintainability, and overall quality of software applications is a crucial aspect of software engineering. This chapter will delve into the fascinating and intricate domain of software quality metrics – the quantitative standards and benchmarks that guide our understanding of the quality of a software system.

Software quality metrics are essential tools that allow stakeholders – developers, testers, managers, and users – to assess a software product's state, identifying its strengths and areas for improvement. They provide an empirical foundation for various processes such as product development, testing, debugging, maintenance, and improvement initiatives. By quantifying specific characteristics of the software, these metrics provide a tangible means to understand the otherwise abstract concept of software quality.

In this section, we will explore several important categories of software quality metrics, including product metrics, process metrics, and project metrics. We'll analyze their significance, methodologies for their calculation, and how they can be effectively utilized to evaluate and enhance software

quality. This exploration will include both static metrics, which are applied to the static software system, and dynamic metrics, which assess the system's behavior during execution.

Software quality metrics not only contribute to the technical soundness of a software system but also help ensure customer satisfaction, profitability, and long-term business success. Therefore, developing an understanding of these metrics is invaluable to anyone involved in the field of software development, from engineers and project managers to executives and software users.

In this section we will examine a few common measures of complexity and maintainability of the class or code. Complex software can be difficult to comprehend. This makes it challenging for developers, particularly new ones, to grasp how different parts of the software interact with each other. This can slow down the onboarding process and development time.

Complex code often leads to higher maintenance efforts. When code is complex, modifications or bug fixes can take longer because it's harder to predict the impacts of changing a single piece of the system. This can result in higher costs over the software's lifecycle.

Complex software tends to be more error prone. Because it's harder to understand, developers are more likely to introduce bugs when making changes. Also, complex code can have many interdependencies, where a change in one area may have unexpected effects elsewhere.

The more complex the software, the more test cases are required to achieve thorough testing. It might also be harder to write these test cases due to the complexity of the logic involved.

Writing simple and maintainable code should be one of our highest priorities. Observing the change in the metric that accompanies our code should aid us in this endeavor. Toward this objective, the first metric that we can (and should) use is the *cyclomatic complexity*.

Cyclomatic Complexity

Cyclomatic Complexity is a metric that quantifies the number of independent paths through a software module. It measures the complexity of decision-making within the code, including loops, conditionals, and branches. A higher cyclomatic complexity value indicates increased complexity and suggests the potential for more bugs and challenges in understanding and maintaining the code.

Within the file `department_visitor.py`, enter the prompt in snippet 6.3 anywhere within this file. Copilot will immediately output the answer.

```
# Question: What is the cyclomatic complexity of the class  
DepartmentStatisticsVisitor?  
# Answer: 1
```

Copilot will tell that the complexity of this class is 1. You may or may not be aware of the meaning of this value. If it is the latter, you can ask Copilot to elaborate.

```
# Question: Is 1 an excellent cyclomatic complexity?  
# Answer: Yes  
  
# Question: Why is one a good value for cyclomatic complexity?  
# Answer: Because it is low
```

Copilot informs us that cyclomatic complexity is good if it is low. Intuitively this makes sense. Code with low complexity means that it is simpler to understand and therefore understand and reason about. It is likely easier to maintain as well. The next metric that we will explore is the *Halstead Complexity measures*.

Halstead Complexity Measures

Halstead complexity measures assess the complexity of a software program based on the number of unique operators and operands used in the code. These measures include metrics such as program length (N_1), program

vocabulary (n1), volume (V), difficulty (D), effort (E), and others. These metrics provide insights into the size and cognitive complexity of the code.

Similar to last time, we will start with a prompt asking Copilot to determine the Halstead Complexity Measure for our visitor class.

```
# Question: What is the Halstead Complexity Measure of the class  
DepartmentStatisticsVisitor?  
# Answer: 2  
  
# Question: What is the Halstead Difficulty Measure of the class  
DepartmentStatisticsVisitor?  
# Answer: 1  
  
# Question: Is 2 a good Halstead Complexity Measure?  
# Answer: Yes  
  
# Question: Is 1 a good Halstead Difficulty Measure?  
# Answer: Yes  
  
# Question: What is a bad Halstead Difficulty Measure?  
# Answer: 10  
  
# Question: What is a bad Halstead Complexity Measure?  
# Answer: 10  
  
# Question: What does a high Halstead Difficulty Measures mean?  
# Answer: It means that the code is hard to understand
```

You may want to continue this Q&A session for a while to see what information can be gleamed from Copilot. Once you are ready to continue, there is one more metric to explore: the *Maintainability index*.

Maintainability Index

The Maintainability Index is a composite metric that combines several factors, including cyclomatic complexity, lines of code, and Halstead complexity measures, to provide an overall measure of software

maintainability. A higher maintainability index suggests easier maintenance and potentially lower complexity.

You should start a similar discussion for the maintainability index in the visitor file.

```
# Question: What is the maintainability index of the class  
DepartmentStatisticsVisitor?
```

```
# Answer: 100
```

```
# Question: Do we want a high Maintainability Index or low  
Maintainability Index?
```

```
# Answer: high
```

```
# Question: Why do we want a high Maintainability Index?
```

```
# Answer: Because it is easier to maintain
```

If we get a low maintainability index, we could refactor to reduce this number. A metric is useful in that it gives us a nail to hang our hat on; that is, we can take that measure and perform some action to improve it. Metrics move us beyond the pure aesthetics or subjectivity of the individual. A metric is real, actionable data. But Copilot has (at least) one more trick up its proverbial sleeve. Copilot is capable of doing more than just writing and assessing our code, it can also address the code's flaws. Let's bug hunt.

6.3 Hunting for bugs

In this section, we will use an elementary (albeit quite contrived) example to demonstrate how we can use Copilot to find and fix the issues in our code. This code is supposed to loop over the list of integers and calculate the sum. However, there is a “blink and you’ll miss it” bug. The sum is assigned the value of *i*, rather than adding the value of *i* to the running total.

Listing 6.11 Simple loop over a list of integers and calculate the sum

```
l = [1, 2, 3, 4, 5]
```

```
if __name__ == '__main__':
    sum = 0
    for i in l:
        sum = i

    print("sum is", sum)
```

To debug this issue, we will introduce a new tool: Copilot Labs. Prior to Copilot Chat, Copilot Labs was the only means by which certain features were available in our IDE, VS Code specifically. For example, we would need to use Copilot Labs to find and fix bugs. The main advantage that Copilot Labs still has today, is that it can access the highlighted contents of your editor pane. This feature allows Copilot Labs to operate directly on the editable code in your IDE. Once you install the extension into your IDE, you should see a Copilot Labs toolkit on the left side of your IDE. Should you need a reminder on how to install an extension into your IDE, please consult appendices A to C for instructions on installing one.

Figure 6.1 The Copilot Labs toolkit menu, which includes options for finding and fixing bugs. The toolkit also provides facilities to enhance your code as well as document it.

▽ BRUSHES



READABLE



ADD TYPES



FIX BUG



DEBUG



CLEAN



LIST STEPS



MAKE ROBUST



CHUNK



DOCUMENT



▽ TEST GENERATION

We shall temporarily change the contents of the main.py file to the code listed in Listing 6.9. Once you have made this change, highlight the code and depress the Fix Bug button within the Copilot Labs toolkit. You should see output like that in Figure 6.2. Copilot Labs was able to determine the issue within this code and provided a suggestion as to how to fix it.

Figure 6.2 Copilot Labs, using the GPT model, has identified the bug and how to address this bug



GitHub Copilot

The bug in this code is that the loop is not summing up the values in the list `l`. Instead, it is overwriting the value of `sum` with each iteration of the loop, so that `sum` ends up being equal to the last value in the list.

To fix this bug, you should change the line `sum = i` to `sum += i`, which will add each value in the list to the running total:

```
l = [1, 2, 3, 4, 5]

if __name__ == '__main__':
    sum = 0
    for i in l:
        sum += i

    print("sum is", sum)
```

With this change, the code will correctly compute the sum of the values in the list and print the result.

Alternatively, we could have copied this code into ChatGPT and asked it to find the bug. However, it is arguable that this is less convenient as you would have to know there was a bug in your code before asking ChatGPT to fix it.

6.4 Covering Code

Code coverage is a measure of how much of your code is being exercised by your tests. It is typically expressed as a percentage and represents the proportion of your code that is executed by your tests.

Code coverage can be used as a metric to evaluate the effectiveness of your tests. If your code coverage is low, it may indicate that parts of your code are not being tested, which could lead to uncaught bugs and other issues.

Alternatively, with high code coverage, you should rest assured that your code is well-tested. This does not guarantee that your code is bug-free, but it should indicate a high degree of confidence that if there are bugs that should be caught in a test.

To determine the code coverage in our Python project, we will use the code coverage tool provided in the `coverage` library `coverage`. The `coverage` library works by instrumenting our code to collect coverage data as it runs. It can collect coverage data for any Python code, including tests, scripts, and modules. By using a code coverage tool like `coverage`, we can better understand how much of our code is being exercised by our tests and identify areas of our code that may need more testing.

First, let's install `coverage` using pip: `pip install coverage`. Next, let's run our tests with `coverage`: `coverage run -m pytest`. This will run your tests and collect coverage data.

Next, we will generate a coverage report. The coverage report will show the code coverage for each file in our project. We create a text-based coverage report using this command: `coverage report` or generate an HTML version of the report using the command: `coverage html`. The HTML version of the

report would be in the htmlcov directory. Figure 6.3 shows the coverage report.

Figure 6.3 The code coverage report showing the coverage for each file in our Information Technology Asset Management system project

Coverage report: 70%

coverage.py v7.2.7, created at 2023-06-07 21:02 -0400

Module	statements	missing	excluded	coverage
itam/__init__.py	4	0	0	100%
itam/domain/__init__.py	12	0	0	100%
itam/domain/address.py	16	8	0	50%
itam/domain/asset.py	54	14	0	74%
itam/domain/budget.py	17	6	0	65%
itam/domain/department.py	7	0	0	100%
itam/domain/depreciation_strategy.py	25	2	0	92%
itam/domain/events/asset_location_updated.py	7	4	0	43%
itam/domain/funding_details.py	18	2	0	89%
itam/domain/hardware.py	34	9	0	74%
itam/domain/location.py	11	0	0	100%
itam/domain/maintenance_schedule.py	20	10	0	50%
itam/domain/organization.py	8	2	0	75%
itam/domain/role.py	10	4	0	60%
itam/domain/software.py	30	14	0	53%
itam/domain/usage_statistics.py	52	27	0	48%
itam/domain/user.py	22	11	0	50%
itam/domain/warranty.py	20	10	0	50%
itam/infrastructure/__init__.py	7	0	0	100%

Code coverage of 70% is a good start. In the real world, we would continue working with our team and generative AI pals to bring this measure up into

the high 90s.

We will transition to a new topic: using generative AI to describe a code listing to us.

6.5 Transliterating code – from code to descriptions

Often one is handed existing source code. Determining the exact entry point of this code, the purpose of the code, and the overall structure of a brownfield project can be challenging. Thankfully, this is one of the areas in which generative AIs truly excel: translating code into textual descriptions.

To begin with, we shall copy the (buggy) code from the previous section into the Copilot Chat dialog box, prefixed with the prompt, “**What does this code do?**”

Figure 6.4 The buggy code from the last section with a prompt asking Copilot Chat to explain this code to us

What does this code do?

```
l=[1, 2, 3, 4, 5]
```

```
if __name__ == '__main__':
    sum = 0
    for i in l:
        sum = i

    print("sum is", sum)
```



What is striking about this explanation is that Copilot Chat detected that there is a bug in the code and offered a suggestion as to how to fix this bug.

Figure 6.5 Copilot Chat explains the code's purpose, identifies the bug, and suggests a fix



GitHub Copilot

This code defines a list `l` containing the integers 1 through 5. It then uses a loop to iterate over the elements of the list and compute their sum. However, there is a bug in the code: the loop is not correctly summing up the values in the list. Instead, it is overwriting the value of `sum` with each iteration of the loop, so that `sum` ends up being equal to the last value in the list.

To fix this bug, the line `sum = i` should be changed to `sum += i`, which will add each value in the list to the running total.

Here's the corrected code:

```
l = [1, 2, 3, 4, 5]

if __name__ == '__main__':
    sum = 0
    for i in l:
        sum += i

    print("sum is", sum)
```

With this change, the code will correctly compute the sum of the values in the list and print the result.

Next, we will ask it something more challenging. We will ask it to explain the behavior test we set up earlier in the chapter. Copy the code into the Copilot Chat window and ask it what this class does. You will likely receive a response similar to the following listing.

Listing 6.12 Copilot Chat explains the behavior test that we wrote previously

The code you provided is not a class but a series of step definitions for a Behavior-Driven Development (BDD) test using the Behave framework in Python. These steps correspond to a specific scenario written in Gherkin language (Given, When, Then format).

These step definitions describe how to interact with an AssetManager class and an InMemoryAssetRepository class. They're meant to test the creation of assets and the calculation of the total cost of all assets.

Let's go through each step:

```
@given('the Asset Manager is running'): This sets up the initial state for the test. It initializes the AssetManager with an InMemoryAssetRepository (which presumably stores assets in memory during the execution of the test) and a mocked AssetLocationMediator.
```

```
@given('the InMemoryAssetRepository is initialized') and @given('the AssetLocationMediator is mocked'): These steps appear to be placeholders, as they don't execute any actions.
```

```
@when('I create an asset with a cost of ${cost}') and @when('I create another asset with a cost of ${cost}'): These steps create Asset instances with the given cost and add them to the AssetManager.
```

```
@then('the total cost of all assets should be ${total_cost}'): This step retrieves all the assets from the AssetManager, calculates the total cost, and checks that it matches the expected total cost.
```

This sequence of steps allows the user to test the process of creating assets and validating the total cost of all assets in the AssetManager.

```

# Question: What does this method do?
# Answer: It gets an asset by its id
async def get_asset(self, asset_id: int):
    asset = self._asset_rest_port.read(asset_id)
    if asset is None:
        raise HTTPException(status_code=404, detail="Asset
not found")
    return from_asset(asset)

```

6.6 Translating from one language to another

As remote as this may seem, you may find yourself looking through a code base written in a programming language with only a passing familiarity. Alternatively, you might find yourself pining for a specific function from another language to be available in your current language. For example, what would the `Asset` and `FundingDetails` classes look like rewritten in JavaScript? Well, let's ask Copilot Chat.

Listing 6.13 Copilot Chat translates our `Asset` and `FundingDetails` classes to JavaScript

```

class Asset {
    constructor(id, name, status, category, cost, useful_life,
salvage_value, purchase_date, locations, funding_details) {
        this.id = id;
        this.name = name;
        this.status = status;
        this.category = category;
        this.cost = cost;
        this.useful_life = useful_life;
        this.salvage_value = salvage_value;
        this.purchase_date = purchase_date;
        this.locations = locations;
        this.funding_details = funding_details;
    }
}

class FundingDetails {
    constructor(asset, funding_source, funding_amount) {
        this.asset = asset;
    }
}

```

```

        this.funding_source = funding_source;
        this.funding_amount = funding_amount;
    }
}

```

We might assume that translating a class or two is too easy. Well, we will attempt something more complex. **What would the AssetManager behavior test look like if written in JavaScript?**

Listing 6.14 Copilot Chat translates our AssetManager behavior test into JavaScript

```

const AssetManager = require('./asset_manager');
const Asset = require('./asset');

describe('AssetManager', () => {
  let assetManager;

  beforeEach(() => {
    assetManager = new AssetManager();
  });

  describe('addAsset', () => {
    it('should add an asset to the asset list', () => {
      const asset = new Asset(1, 'Computer', 'Active',
      'Hardware', 1000.0, 5, 100.0, new Date(), [], null);
      assetManager.addAsset(asset);
      expect(assetManager.assets.length).toBe(1);
      expect(assetManager.assets[0]).toBe(asset);
    });
  });

  describe('removeAsset', () => {
    it('should remove an asset from the asset list', () => {
      const asset = new Asset(1, 'Computer', 'Active',
      'Hardware', 1000.0, 5, 100.0, new Date(), [], null);
      assetManager.addAsset(asset);
      assetManager.removeAsset(asset);
      expect(assetManager.assets.length).toBe(0);
    });
  });
});

```

```

describe('getAssetById', () => {
  it('should return the asset with the given ID', () => {
    const asset1 = new Asset(1, 'Computer', 'Active',
'Hardware', 1000.0, 5, 100.0, new Date(), [], null);
    const asset2 = new Asset(2, 'Printer', 'Active',
'Hardware', 500.0, 3, 50.0, new Date(), [], null);
    assetManager.addAsset(asset1);
    assetManager.addAsset(asset2);
    const result = assetManager.getAssetById(2);
    expect(result).toBe(asset2);
  });

  it('should return null if no asset with the given ID is
found', () => {
    const result = assetManager.getAssetById(1);
    expect(result).toBeNull();
  });
});
});

```

Additionally, you might wish to have GPT rewrite the code in idiomatic JavaScript. You need only say: “Write AssetManager in idiomatic JavaScript.” Copilot Chat will inform you that it will define the AssetManager class using the class syntax introduced in ES6.

Listing 6.15 The AssetManager classe rewritten in idiomatic JavaScript

```

class AssetManager {
  constructor() {
    this.assets = [];
  }

  addAsset(asset) {
    this.assets.push(asset);
  }

  removeAsset(asset) {
    const index = this.assets.indexOf(asset);
    if (index !== -1) {
      this.assets.splice(index, 1);
    }
  }
}

```

```

        getAssetById(id) {
            return this.assets.find(asset => asset.id === id) || null;
        }
    }
}

```

Here's an example of what the AssetManager behavior test might look like in JavaScript using the Jest testing framework (which Copilot Chat included without our suggestion):

Listing 6.16 Copilot Chat rewrites our AssetManager test in idiomatic JavaScript

```

const AssetManager = require('./asset_manager');
const Asset = require('./asset');
describe('AssetManager', () => {
    let assetManager;
    beforeEach(() => {
        assetManager = new AssetManager();
    });
    describe('addAsset', () => {
        it('should add an asset to the asset list', () => {
            const asset = new Asset(1, 'Computer', 'Active',
'Hardware', 1000.0, 5, 100.0, new Date(), [], null);
            assetManager.addAsset(asset);
            expect(assetManager.assets.length).toBe(1);
            expect(assetManager.assets[0]).toBe(asset);
        });
    });
    describe('removeAsset', () => {
        it('should remove an asset from the asset list', () => {
            const asset = new Asset(1, 'Computer', 'Active',
'Hardware', 1000.0, 5, 100.0, new Date(), [], null);
            assetManager.addAsset(asset);
            assetManager.removeAsset(asset);
            expect(assetManager.assets.length).toBe(0);
        });
    });
    describe('getAssetById', () => {
        it('should return the asset with the given ID', () => {
            const asset1 = new Asset(1, 'Computer', 'Active',
'Hardware', 1000.0, 5, 100.0, new Date(), [], null);
            const asset2 = new Asset(2, 'Printer', 'Active',
'Hardware', 500.0, 3, 50.0, new Date(), [], null);
            assetManager.addAsset(asset1);
            assetManager.addAsset(asset2);
            expect(assetManager.getAssetById(1)).toBe(asset1);
            expect(assetManager.getAssetById(2)).toBe(asset2);
        });
    });
}

```

```

        assetManager.addAsset(asset1);
        assetManager.addAsset(asset2);
        const result = assetManager.getAssetById(2);
        expect(result).toBe(asset2);
    });
    it('should return null if no asset with the given ID is
found', () => {
    const result = assetManager.getAssetById(1);
    expect(result).toBeNull();
});
});
});
);

```

In this example, we use the Jest testing framework to define a set of behavior tests for the `AssetManager` class. We are using the `describe` library to determine test suites and individual tests.

In each test, we create an instance of the `AssetManager` class and perform some operation on it, such as adding or removing an asset. We are then using the `expect` function to make assertions about the behavior of the `AssetManager` class, such as checking that an asset was added to the asset list or that an asset was removed from the asset list.

Note that the syntax for defining classes and constructors is slightly different in JavaScript compared to Python, and we are using the required function to import the `AssetManager` and `Asset` classes from their respective modules.

Finally, for the big finale: We will start a new ChatGPT 4.0 session and paste in our Mermaid diagram from Chapter 3. Lead with the prompt, **“Here is a mermaid diagram of my application. Please implement in Golang.”** Marvel as it returns the code, rendered in Golang.

Listing 6.17 Copilot Chat rewrites our Asset class in idiomatic Golang

```

type Asset struct {
    id          int
    name        string
    status      string
    category    string
}

```

```

        fundingDetails *FundingDetails
    }

func NewAsset(name, status, category string, fundingDetails
 *FundingDetails) *Asset {
    return &Asset{
        name:           name,
        status:         status,
        category:       category,
        fundingDetails: fundingDetails,
    }
}

```

Next, tell it to switch to Java.

Listing 6.18 Copilot Chat rewrites our Asset class in idiomatic Java

```

class Asset {
    private int id;
    private String name;
    private String status;
    private String category;
    private FundingDetails fundingDetails;

    public Asset(String name, String status, String category,
FundingDetails fundingDetails) {
        this.name = name;
        this.status = status;
        this.category = category;
        this.fundingDetails = fundingDetails;
    }
}

```

6.7 Summary

- Unit testing: Focuses on testing individual components or units of code to identify bugs and issues within specific units. Unit tests will be the most numerous in your codebase.
- Integration testing: Tests the interaction between different components or modules of the software to ensure seamless integration and detect

communication problems.

- Behavior testing: Tests a system's functionality from an end user's perspective, ensuring it meets requirements and specifications.
- Mock objects: Simulate the behavior of natural objects in a controlled way, useful for testing and simulating error conditions. Mock objects are especially good at mimicking parts of the system which are needed for the test to run, but outside the scope of the test. For example, if your class had a constructor argument for a database, but you do not want to test the database directly, since the data may change, causing your test to be inconclusive, non-repeatable, or non-deterministic.
- Cyclomatic Complexity: Measures the number of independent paths through a software module, indicating complexity and potential for bugs.
- Halstead complexity measures: Assess software complexity based on unique operators and operands, providing insights into code size and cognitive complexity.
- Maintainability Index: Combines factors like cyclomatic complexity, lines of code, and Halstead measures to evaluate software maintainability.
- Code coverage: Metric for evaluating test effectiveness, indicating the extent to which code is tested and the potential for uncaught bugs. Generally, higher is better.
- Language familiarity: Needing to navigate code in an unfamiliar programming language or desiring features from another language in the current one.

7 Coding Infrastructure and Managing Deployments

This chapter covers

- Creating a Dockerfile with the assistance of Copilot
- Drafting your infrastructure as code using Large Language Models
- Managing Docker images with a container registry
- Harnessing the power of Kubernetes
- Releasing your code effortlessly using GitHub Actions

There is nothing more demoralizing than having an application sit unused. For this reason, fast-tracking a well-tested application to production is the stated goal of every competent developer. Since we spent the last chapter testing our product, it is now ready for launch.

This chapter will focus on that pivotal moment of transitioning from development to product launch. During this critical phase understanding deployment strategies and best practices becomes essential to ensuring a successful product launch.

With our application successfully secured and tested, it's time to shift our attention toward launching the product. To this end, we will leverage the powerful capabilities of Large Language Models (LLMs) to explore various deployment options tailored to cloud infrastructure.

By harnessing the power of LLMs and embracing their deployment options and methodologies, we can confidently navigate the complex landscape of launching our product, delivering a robust and scalable solution to our customers while leveraging the benefits of cloud computing.

First, we will develop deployment files for Docker. We will explore how to create Docker images and define deployment files. Additionally, we will

discuss best practices for containerizing our application and achieving seamless deployment.

Next, we will leverage Terraform to define our infrastructure as code and automate the deployment of Elastic Compute Cloud (EC2) instances on AWS. We will demonstrate how to write Terraform scripts to provision and deploy our application on EC2 instances, ensuring consistent and reproducible infrastructure setups.

Then, we will utilize LLMs to deploy our application onto Kubernetes (AWS EKS/ECS). We will have GitHub Copilot create the appropriate Kubernetes deployment files to streamline our deployment process and efficiently manage our application's lifecycle. Given the relative simplicity of our application, we will not need a Kubernetes package manager such as Helm. However, as the complexities and dependencies of services grow, you may want to explore it as one option. Thankfully, Copilot can write Helm charts for you as well!

Bottom of FormLastly, we will briefly showcase migrating from local to automated deployments using GitHub Actions. We can automate our build and deployment processes by integrating LLMs with this widespread continuous integration and delivery (CI/CD) tool, ensuring faster and more efficient deployments.

NOTE

While this chapter will use AWS as our cloud provider, the principles and practices covered in this chapter can be adapted and applied to other cloud platforms and even on-premises infrastructure without virtualization (bare metal), allowing us to adapt and scale our product deployment strategy as our business needs evolve. You will find that by employing LLMs and using infrastructure as code, you can (partially) mitigate the vendor lock-in that is very common to cloud platforms.

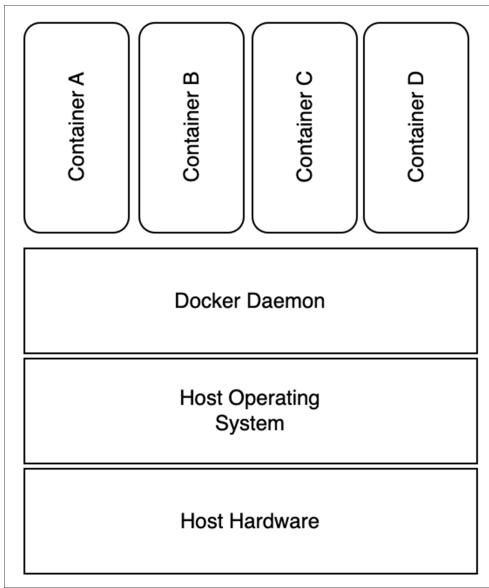
You should note that if you choose to deploy this (or any application) to AWS, then there will be a cost associated with your activity. AWS and most

cloud providers give you free trials to learn their platforms (Google Cloud Platform and Azure for example), but once those credits have expired, you might get hit with a rather unexpectedly large bill. If you decide to follow along in this chapter, you should set threshold alerts for an amount you can comfortably afford. Section 1.9 of Andreas Wittig and Michael Wittig's *Amazon Web Services in Action, Third Edition*, March 2023, Manning Publishing, is an excellent resource for setting up such a billing notification alert.

7.1 Building a Docker image and “deploying” it locally

As you may remember from Chapter 6, Docker is a containerization platform that allows one to run applications with little or no installation of an application (outside of Docker) in the traditional sense. Unlike a virtual machine, which simulates an entire operating system, a container shares the host system's kernel (the core part of the operating system) and uses the host system's operating system's capabilities, while isolating the application processes and file systems from the host. This allows you to run multiple isolated applications on a single host system, each with its own environment and resource limits. The following diagram should give you a sense of the relationship between the Docker runtime and the host.

Figure 7.1 Docker makes use of the host's operating system while isolating each of the containers. This makes Docker containers lightweight compared to virtual machines, as they do not require a full OS to run.



One of the more exciting features, from a production readiness perspective, is that Docker makes it easier to run applications that can self-heal in some sense. If they fail or fall over at runtime, you can configure them to restart without intervention. In this section, we will use Copilot to create the file (called a *Dockerfile*) from which we will build our *Docker image*.

Docker Images

Docker images are like blueprints for Docker containers. They are portable, including all the dependencies (libraries, environment variables, code, etc.) required for the application to run.

Running Docker instances are called Docker containers. Given their lightweight nature, you can run multiple containers on a single host without issue. We can do this because the containerization technology shares the OS kernel, operating in an isolated user space.

NOTE

Originally, your author wanted to use AWS CodeWhisperer as the LLM for this chapter. It seemed logical, given the intended cloud platform. However, at the time of this writing, AWS CodeWhisperer only supports

programming in a programming language. It does not have facilities for infrastructure as code.

We will begin with the following prompt to have Copilot draft the Dockerfile for us.

```
# Create a Dockerfile for this Python app. The main class is
main.py. Use Python 3.10 and install
# the dependencies using the requirements.txt file in this
directory. The app should run on port 8080.
```

You might be left with an empty file (other than this comment). Support for infrastructure as code is ever-evolving (not unlike the LLM ecosystem, in general). According to Copilot Chat, GitHub Copilot is capable of creating a Dockerfile for you; however, you will have to goad it with the following steps:

1. In the Dockerfile, type `FROM python:` and wait for Copilot to suggest a version of Python to use. Select the version you want to use.
2. Type `WORKDIR /app` to set the working directory for the container.
3. Type `COPY . /app` to copy the contents of your project into the container.
4. Type `RUN pip install --trusted-host pypi.python.org -r requirements.txt` to install the dependencies for your project.
5. Type `EXPOSE 8080` to expose port 8080 for the container.
6. Type `CMD ["python", "main.py"]` to specify the command to run when the container starts.

Alternatively, you might want to copy and paste the same prompt that you had previously written into the Dockerfile into Copilot Chat prompt window. Copilot Chat will give you the desired content for the Dockerfile.

Listing 7.1 The contents of the Dockerfile to build the Docker image and prepare it for its runtime lifecycle

```
FROM python:3.10-slim-buster
WORKDIR /app
COPY . /app
RUN pip install --trusted-host pypi.python.org -r requirements.txt
EXPOSE 8080
CMD ["python", "main.py"]
```

With a Dockerfile, we will build an image for deploying and running our application. We can enter the following command to build our application (run from the directory where the Dockerfile lives, and do not forget the trailing dot). You will need internet access to download the dependencies and create the image.

```
docker build -t itam:latest .
```

Building a Docker image can run for a few seconds to a few minutes, depending on which images and packages are installed on your system and your Internet connection speed. Your patience will soon be rewarded, as you will shortly have an application you can install nearly anywhere from the lowliest commodity hardware to the most oversized hardware offered by your favorite cloud provider. Before running it anywhere, however, you should try to get it running locally. Should you forget the command, Copilot Chat will happily and helpfully assist.

```
docker run -p 8000:8000 -d --name itam itam:latest
```

You can confirm that your Docker container is running by issuing this command at the command line: `docker ps | grep itam`. You should see the running instance.

7.2 Standing up infrastructure by Copiloting Terraform

Using a Docker image on your computer is useful when creating and testing an app. But when it comes time to launch your application, we will need a machine with a little more heft than our local computers. In this section, we'll use GitHub Copilot to help us set up and control our AWS

infrastructure by having Copilot write the requisite deployment descriptors for an infrastructure-as-code tool called Terraform. Terraform is made by HashiCorp and lets us write what we want our infrastructure to look like using a domain-specific language (DSL). This DSL saves us from understanding all the complexities and intricacies that each cloud service provider uses to provision hardware. Additionally, it allows us to store and version our infrastructure using infrastructure as code.

To start, we want to create a file called `ec2.tf`, add the prompt to inform Copilot that we intend for this to be a Terraform file and how we want our infrastructure stood up. You may notice that Copilot needs us to enter the first word of a given line before it can be cajoled to continue.

Listing 7.2 An example of a Terraform file, including instance size and instructions on how it should be built and configured

```
# Create a Terraform file that provisions an AWS EC2 instance
# of type t2.micro, installs the Docker daemon, and returns the
# instance's hostname.

provider "aws" {
    region = "us-west-2"
}

resource "aws_instance" "example" {
    ami          = "ami-0c55b159cbfafe1f0"
    instance_type = "t2.micro"
}
```

You may find that Copilot skipped over one small but crucial detail: Copilot did not provide code for installing and provisioning Docker. Given that Docker is required for running our application, we will need to correct this oversight. In fact, you may need to update the file manually to include the command to install Docker, thusly:

```
resource "aws_instance" "example" {
    ami          = "ami-0c55b159cbfafe1f0"
    instance_type = "t2.micro"
```

```

    user_data = <<-EOF
        #!/bin/bash
        sudo yum update -y
        sudo yum install -y docker
        sudo service docker start
        sudo usermod -a -G docker ec2-user
        sudo docker run -d -p 80:80 nginx
    EOF
}

```

Copilot should produce a complete Terraform file that resembles the content of Listing 7.3. Likely, the code is not exactly matching this Listing, but that should be fine so long as it contains the key features: namely, the provider, the instance, the script to add the Docker daemon, the keypair, and the security group.

Listing 7.3 Complete listing the Terraform file to create the smallest EC2 instance available

```

# Create a Terraform file that provisions an AWS EC2 instance
# of type t2.micro, installs the Docker daemon, and returns the
# hostname of the instance.

provider "aws" {
    region = "us-west-2"
}

resource "aws_instance" "example" {
    ami           = "ami-0c55b159cbfafe1f0"
    instance_type = "t2.micro"

    user_data = <<-EOF
        #!/bin/bash
        sudo yum update -y
        sudo yum install -y docker
        sudo service docker start
        sudo usermod -a -G docker ec2-user
        sudo docker run -d -p 80:80 nginx
    EOF
}

connection {
    type     = "ssh"
    user     = "ec2-user"
}

```

```

    private_key = file("~/ssh/id_rsa")
    host        = self.public_ip
}

lifecycle {
    create_before_destroy = true
}

depends_on = [aws_security_group.allow_http]
}

resource "aws_security_group" "allow_http" {
    name          = "allow_http"
    description   = "Allow HTTP inbound traffic"
    vpc_id        = "vpc-12345678"

    ingress {
        description = "HTTP from VPC"
        from_port   = 80
        to_port     = 80
        protocol    = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }
}

output "public_dns" {
    value = aws_instance.example.public_dns
}

```

If you are using the default Virtual Private Cloud (VPC), then the `vpc_id` entry on line 35 is not strictly necessary. While you will find many of the default configurations and conventions chosen by the AWS team to make sense, if you have more stricter security requirements, or if you know everything about your infrastructure and assume nothing, then you might consider setting up a new VPC from scratch, using Terraform. You will want to change the key pair entry on line 21 to be a key pair to which you have access.

Once you have completed this file satisfactorily, you should run the `terraform init` command. The `terraform init` command initializes a new or existing Terraform working directory. This command downloads

and installs the required provider plugins and modules specified in your configuration files. This command gets everything ready to go.

Next, you will have Terraform explain the changes that it intends to make. You do this with the `terraform plan` command. The `terraform plan` command creates an execution plan for your infrastructure changes. This command shows you what changes Terraform will make to your infrastructure when you apply your configuration files. The plan will show you which resources will be created, modified, or destroyed, and any other changes that will be made to your infrastructure.

NOTE

You may get the error when running `terraform plan` for the first time “Error: configuring Terraform AWS Provider: no valid credential sources for Terraform AWS Provider found.” You get this error because Terraform attempts to connect to AWS but cannot supply AWS with proper credentials. To address this issue, you will need to create (or edit) the file called `~/.aws/credentials` and add your ITAM AWS Access Key ID and AWS Secret Access Key credentials. More details can be found in section **4.2.2 Configuring the CLI** of *Amazon Web Services in Action, Third Edition* for full details on how to accomplish this correctly.

Finally, to apply the Terraform changes, you would use the `terraform apply` command. Terraform will then read the configuration files in the current directory and apply any changes to your infrastructure. If you have made any changes to your configuration files since the last time you ran `terraform apply`, for example, if we need to start up a new database instance or change the size of your EC2, then Terraform will show you a preview of the changes that will be made and prompt you to confirm before applying the changes.

Were you to apply these changes, then in a manner of minutes, you would have a brand new EC2 instance running in your VPC. However, this is only one-half of the equation. Having computing power at your fingertips is fantastic; however, you need something to apply this power. We could use

this EC2 to run our Information System Asset Management system in this case. The following section will briefly demonstrate transferring a locally built image to another machine.

7.3 Moving a Docker Image around (the hard way)

First, we will export a Docker image from our local machines and load it onto a remote machine. We will use the commands `docker save` and `load` to accomplish this. You can use the `docker save` command on your local machine to save the image to a tar archive. The following command will save the image to a tar archive named `<image-name>.tar`: `docker save -o <image-name>.tar <image-name>:<tag>`.

Next, you can transfer the tar archive to the remote machine using a file transfer protocol such as SCP or SFTP. You can use the `docker load` command on the remote machine to load the image from the tar archive: `docker load -i <image-name>.tar`. This will load the image into the local Docker image cache on the remote machine. Once the image has been loaded, you may use the `docker run` command to start the image and run the Docker container, as you did after you built it. Further, you could then add this image to your Docker compose file, in which you have the Postgres database and Kafka instances.

NOTE

The previous treatment of Terraform is heavily abridged. When you are ready to get serious with Terraform, your go-to resource should be Scott Winkler's *Terraform in Action* (May 2021, Manning Publishing).

This section examined how to package up our images and load them on remote hosts. While this is easily scriptable, with the advent of container registries, it is now easier than ever to manage deployments without slinging them all around the internet. In the next section, we will explore one such tool Amazon's Elastic Container Registry (ECR).

7.4 Moving a Docker Image around (the easy way)

Docker images, the blueprints for our containers, are a fundamental building block of containerized applications. Managing them correctly ensures that we maintain clean, efficient, and organized development and deployment workflows. Amazon ECR serves as a fully-managed Docker container registry that makes it easy for developers to store, manage, and deploy Docker container images.

First, let's dive into pushing Docker images to ECR. This process is vital to making your images accessible for use and deployment. We'll walk through setting up your local environment, authenticating with ECR, and pushing your image. Before we can move our image to ECR, we must create a repository to house said image. This can be done from the AWS Management Console or as we will do shortly, use the AWS Command Line Interface (CLI). The command to create a new repository for our image is: `aws ecr create-repository --repository-name itam`

Next, we will want to tag our Docker image with the ECR repository URL and the image name. We may want to call it latest or use semantic versioning. Tagging will allow us to easily roll back or forward versions of our system. We would tag our application image to latest using the following command: `docker tag itam:latest 123456789012.dkr.ecr.us-west-2.amazonaws.com/itam:latest`.

Then, we will need to authenticate Docker to the ECR registry using the `aws ecr get-login-password` command. This will generate a Docker login command that you can use to authenticate Docker to the registry. The command to login is `aws ecr get-login-password --region us-west-2 | docker login --username AWS --password-stdin 123456789012.dkr.ecr.us-west-2.amazonaws.com`

Finally, we will push the Docker image to the ECR registry using the `docker push` command. We would do that thusly: `docker push 123456789012.dkr.ecr.us-west-2.amazonaws.com/itam:latest`

Once our image is in our registry, our deployment options have greatly increased. We could, for example, write a bash script that will log on to our EC2 instance and perform a docker pull to download and run the image on that EC2. Alternately, we may want to adopt a more bullet-proof deployment pattern. In the next section, we're going to walk through the process of setting up and launching our application on a powerful cloud service called Elastic Kubernetes Service (EKS). EKS is a managed Kubernetes service provided by AWS (Amazon Web Services). Let's dive in!

Top of Form
Bottom of Form

7.5 Deploying our application on to AWS Elastic Kubernetes Service (EKS)

Kubernetes confers many benefits over simply running Docker images on EC2 instances. For one, managing and scaling our application becomes considerably more straightforward with Kubernetes. Also, with Kubernetes, we do not have to spend a lot of additional time thinking about what our infrastructure should look like. Plus, thanks to its automatic management of the lifecycles of its images, known as pods, our application will essentially be self-healing. This means that if something goes wrong, Kubernetes can automatically fix it, keeping our application running smoothly at all times.

First, we will need a deployment descriptor written in YAML (Yet Another Markup Language or YAML Ain't Markup Language, depending on who you ask), which will describe the state we want our Information Technology Asset Management system to be in at all times. This file (typically called deployment.yaml) will provide the template against which Kubernetes will compare the current, running system against and make corrections as needed.

Listing 7.4 A Kubernetes deployment file for the Information Technology Asset Management system

```
# Create a Kubernetes deployment file for the item application.  
The image name is item:latest
```

```
# The deployment will run on port 8000

apiVersion: apps/v1
kind: Deployment
metadata:
  name: itam-deployment
  labels:
    app: itam
spec:
  replicas: 1
  selector:
    matchLabels:
      app: itam
  template:
    metadata:
      labels:
        app: itam
    spec:
      containers:
        - name: itam
          image: itam:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 8000
```

This will not work, however. Kubernetes will not be able to find the image that we reference in the deployment descriptor file. To correct this, we will need to tell Kubernetes to use our newly minted ECR. Thankfully, this is not as challenging as it may sound. We just have to update the image entry in our file to point to the ECR image, as well as grant EKS permissions to access ECR (okay maybe it is a little trickier, but it is manageable).

First, update the deployment yaml to use the ECR image. It would resemble this: `image: 123456789012.dkr.ecr.us-west-2.amazonaws.com/itam:latest`. Then, we would need to define a policy for EKS to use. We would then apply the policy using either the AWS CLI or the IAM Management Console. While applying the policy is (slightly) outside of the scope of this book, you could use Copilot to define this policy. The resulting policy would resemble the following listing.

Listing 7.5 An IAM policy to allow EKS to pull images from ECR

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "AllowPull",  
            "Effect": "Allow",  
            "Principal": {  
                "AWS": "arn:aws:iam::<aws_account_id>:role/<role>"  
            },  
            "Action": [  
                "ecr:GetDownloadUrlForLayer",  
                "ecr:BatchGetImage",  
                "ecr:BatchCheckLayerAvailability"  
            ],  
            "Resource": "arn:aws:ecr:<region>:  
<aws_account_id>:repository/<repository_name>"  
        }  
    ]  
}
```

Once the EKS can pull down the image from ECR, you will see a Pod start to run. However, you would not have any way to access this Pod externally. We would need to create a Service. In Kubernetes, a Service is an abstraction that defines a logical set of Pods (the smallest and simplest unit in the Kubernetes object model that you create or deploy) and a policy to access them.

Services enable communication between different parts of an application and between different applications. They help distribute network traffic and load balance by exposing the Pods to the network and to other Pods within Kubernetes.

Listing 7.6 A Kubernetes services file to enable external access for our application

```
# Please create a service for the application that uses a load  
balancer type egress
```

```
apiVersion: v1
kind: Service
metadata:
  name: itam-service
spec:
  type: LoadBalancer
  selector:
    app: itam
  ports:
    - name: http
      port: 80
      targetPort: 8000
```

Kubernetes is responsible for routing all requests from this ingress through the service to the running pods, irrespective of what host they are running on. This allows for seamless failover. Kubernetes expects things to fail. It banks on it. As a result, many of the best practices in distributed systems are baked into Kubernetes. Getting to Kube is a significant first step to having a reliable, highly available system. In the next section, we will examine how to ease the burden of getting our application onto Kubernetes repeatably and continuously. We will look at building out a small deployment pipeline using GitHub actions.

7.6 Setting up a Continuous Integration/Continuous Deployment pipeline in GitHub Actions

If releasing is hard, then it will not be done often. This limits our ability to add value to the application and thus to our stakeholders. However, automating the deployment process significantly reduces the time to release. This allows for more frequent releases, accelerating the pace of development and enabling faster delivery of features to users. Continuous Integration/Continuous Deployment (CI/CD) pipelines limit the risk associated with deployment. By making smaller, more frequent updates, any issues that arise can be isolated and fixed quickly, minimizing the potential impact on the end users. These pipelines facilitate seamless integration of code changes and expedite deployment, simplifying the software release process.

GitHub Actions allows us to construct customized CI/CD pipelines directly within our GitHub repositories. This makes the development workflow more efficient and enables the automation of various steps, freeing us to focus on coding rather than the logistics of integration and deployment.

This section provides a concise introduction to setting up a CI/CD pipeline using GitHub Actions and GitHub Copilot. Please note that this will not be a comprehensive guide but a survey that introduces the potential benefits and general workflow. This should serve as a primer, giving you an insight into how these tools can be used to optimize your software development process.

First, we will create a file in our project in the path: `.github/workflows`. Note the leading dot. We can call this file `itam.yaml` or whatever you desire. On the first line of this file, add the following prompt: # Create a GitHub Actions workflow that builds the ITAM application on every merge to the main branch and deploys it to EKS.

NOTE

: Like many of the infrastructure-related tasks that we have put to Copilot in this chapter, Copilot needs a lot of assistance in creating this file for us. We need to be aware of the structure of this file and how to begin every line. It would make sense in cases such as this one to ask ChatGPT or Copilot Chat to build the file for us.

Listing 7.7 The beginning of our GitHub Actions file, which we use to build and deploy our application

```
# Create a GitHub Actions workflow that builds the ITAM
application on every merge to the main branch and deploys it to
EKS.
```

```
name: Build and Deploy to EKS
```

```
on:
```

```
  push:
```

```
branches:  
  - main  
jobs:
```

The build job will first check out the code from our GitHub repository. It uses the code written in the module actions/checkout version 2. Similarly, it will next, grab the EKS command line interface and configure the credentials to connect to EKS. You will note that the AWS access key and secret are values that are automatically passed into the application. GitHub Actions uses a built-in secret management system to store sensitive data such as API keys, passwords, or certificates. This system is integrated into the GitHub platform and allows you to add, remove, or update secrets (and other sensitive data) at both the repository and organization level. Secrets are encrypted before they're stored and are not shown in logs or available for download. They're only exposed as environment variables to the GitHub Actions runner, making it a secure way to handle sensitive data.

Likewise, you can create environmental parameters and use them in your Actions. For example, look at the variable ECR_REGISTRY. This variable is created using the output from the login-ecr function. In this case, you would still need to hardcode the Elastic Container Registry (ECR) in your Actions file. However, you would want to do this because of consistency and the need to only manage it in one place in the file. Most of these steps should seem familiar to you as we have used them throughout the chapter. That is the magic of automation: it does it for you.

Listing 7.8 The build and deploy steps of our GitHub Actions file

```
build:  
  runs-on: ubuntu-latest  
  
steps:  
  - name: Checkout code  
    uses: actions/checkout@v2  
  
  - name: Set up EKS CLI  
    uses: aws-actions/amazon-eks-cli@v0.1.0
```

```

- name: Configure AWS credentials
  uses: aws-actions/configure-aws-credentials@v1
  with:
    aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
    aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    aws-region: us-west-2

- name: Build and push Docker image
  env:
    ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
    ECR_REPOSITORY: itam
    IMAGE_TAG: ${{ github.sha }}
  run: |
    docker build -t
    $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
    docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG

- name: Deploy to EKS
  env:
    ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
    ECR_REPOSITORY: itam
    IMAGE_TAG: ${{ github.sha }}
  run: |
    envsubst < k8s/deployment.yaml | kubectl apply -f -
    envsubst < k8s/service.yaml | kubectl apply -f -

```

The final part of the file logs into AWS ECR. The steps in the Actions file invoke this section. Upon completion, it will return the output to the calling function.

Listing 7.9 A beginning of our GitHub Actions file to build and deploy our application to EKS

```

login-ecr:
  runs-on: ubuntu-latest
  steps:
    - name: Login to Amazon ECR
      id: login-ecr
      uses: aws-actions/amazon-ecr-login@v1
      with:
        registry: <your-ecr-registry>
        aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}

```

```
aws-secret-access-key: ${{  
secrets.AWS_SECRET_ACCESS_KEY }}}
```

In conclusion, exploring Code-as-Infrastructure has enabled us to understand its vital role in any project and how it can be better managed through code. Tools like Terraform provide streamlined solutions for managing infrastructure, while GitHub's code-centric features aid in maintaining the overall workflow.

Introducing Continuous Integration and Continuous Deployment (CI/CD) pipelines, primarily through platforms like GitHub Actions, highlights the importance of automating the software delivery process. Automating such processes increases the speed and reliability of the software development life cycle and minimizes the chances of human errors.

The journey of managing infrastructure as code is ever-evolving, with new tools and practices emerging. It requires a constant learning and adaptation mindset. This chapter has given you a glimpse of the benefits and possibilities.

7.7 Summary

- The transition from application development to product launch:
Discusses the process from when an application is ready for production to when it is made live.
- Deployment strategies and best practices for cloud infrastructure:
Details on how to set up your cloud infrastructure and the recommended guidelines to achieve optimal performance.
- Use of Docker for containerizing applications: It explains how Docker can bundle an application and its dependencies into a single object, making it easy to deploy and manage.
- Introduction to Terraform for infrastructure as code: Describes how Terraform enables you to use code to manage your infrastructure, thereby increasing efficiency and reducing errors.
- Management of application deployment via Kubernetes: Discuss how Kubernetes simplifies containerized applications' deployment, scaling,

and management.

- Adaptation of methods to different cloud platforms or on-premises infrastructure: Explain how the methods discussed in the book can be adapted for use on different cloud platforms or on-premises deployments.
- GitHub Copilot's role in creating Dockerfiles and Terraform files: Discusses how GitHub Copilot, an AI-powered code assistant, can help you create Docker and Terraform files more efficiently and accurately.
- Exporting Docker image from local to remote machine: Explains the steps to move a Docker image from your local machine to a remote server.
- Deployment on AWS's Elastic Kubernetes Service (EKS): Discusses how to deploy a containerized application on AWS EKS, a managed Kubernetes service.
- Creation of Kubernetes YAML deployment descriptors: Details how to write a Kubernetes deployment descriptor in YAML format, which describes the desired state for your deployment.
- Formation of Kubernetes services for network traffic distribution and communication: Explain how to create Kubernetes services, which abstract how you communicate with and route traffic to your pods.
- Pushing Docker image to Amazon's Elastic Container Registry (ECR): Describes how to upload your Docker images to ECR, a fully managed Docker container registry provided by AWS.
- Migration from local to automated deployments using GitHub Actions: Discusses how to automate your deployment process using GitHub Actions, a CI/CD platform.

welcome

Thank you for securing your copy of the MEAP for *AI-powered Developer*.

This book is designed to transition readers from having little to no experience with AI-powered coding tools to becoming proficient developers able to leverage these powerful technologies in their projects. Given the broadness of introducing both programming and AI concepts, we assume you're familiar with Python.

My journey to writing this book has been enriched with diverse experiences. Working with various projects, I've faced the task of integrating and leveraging AI tools in development workflows. Back then, the robustness of AI assistance and documentation we enjoy today was not available, making the transition challenging yet incredibly rewarding.

Throughout this book, we'll delve into the core aspects of generative AI models like ChatGPT and GitHub Copilot, providing a hands-on understanding of their application in real-world scenarios. We'll explore how these AI models can interact with Python to create efficient, self-documenting code, make predictive suggestions, and assist in the design phase.

It's essential for professionals to thoroughly understand their tools, and this book ensures just that. We will familiarize you with various features of AI models, ranging from beginner-friendly to advanced capabilities. Additionally, we will also discuss how these AI models fit into the testing cycle, their role in debugging, and their ability to explain code and concepts.

The integration of AI tools like ChatGPT and GitHub Copilot into coding has introduced revolutionary changes and possibilities in the software development community. *AI-powered Developer* aims to empower you to take full advantage of these opportunities, covering a broad range of topics

to ensure that you can create the applications you envision while, or even after, reading the book.

Your feedback is crucial for crafting the best possible book, so I encourage you to share your comments and questions in the [liveBook Discussion forum](#). I'm eager to know if my explanations are resonating with you and aiding in your understanding!

-Nathan B Crocker

In this book

[welcome](#) [1 Understanding Large Language Models](#) [2 Getting Started with Generative AI](#) [3 Designing Software with ChatGPT](#) [4 Building Software with GitHub Copilot](#) [5 Managing Data with GitHub Copilot and Copilot Chat](#) [6 Testing, Assessing, and Explaining with Large Language Models](#) [7 Coding Infrastructure and Managing Deployments](#)

welcome

Thank you for securing your copy of the MEAP for *AI-powered Developer*.

This book is designed to transition readers from having little to no experience with AI-powered coding tools to becoming proficient developers able to leverage these powerful technologies in their projects. Given the broadness of introducing both programming and AI concepts, we assume you're familiar with Python.

My journey to writing this book has been enriched with diverse experiences. Working with various projects, I've faced the task of integrating and leveraging AI tools in development workflows. Back then, the robustness of AI assistance and documentation we enjoy today was not available, making the transition challenging yet incredibly rewarding.

Throughout this book, we'll delve into the core aspects of generative AI models like ChatGPT and GitHub Copilot, providing a hands-on understanding of their application in real-world scenarios. We'll explore how these AI models can interact with Python to create efficient, self-documenting code, make predictive suggestions, and assist in the design phase.

It's essential for professionals to thoroughly understand their tools, and this book ensures just that. We will familiarize you with various features of AI models, ranging from beginner-friendly to advanced capabilities. Additionally, we will also discuss how these AI models fit into the testing cycle, their role in debugging, and their ability to explain code and concepts.

The integration of AI tools like ChatGPT and GitHub Copilot into coding has introduced revolutionary changes and possibilities in the software development community. *AI-powered Developer* aims to empower you to take full advantage of these opportunities, covering a broad range of topics

to ensure that you can create the applications you envision while, or even after, reading the book.

Your feedback is crucial for crafting the best possible book, so I encourage you to share your comments and questions in the [liveBook Discussion forum](#). I'm eager to know if my explanations are resonating with you and aiding in your understanding!

-Nathan B Crocker

In this book

[welcome](#) [1 Understanding Large Language Models](#) [2 Getting Started with Generative AI](#) [3 Designing Software with ChatGPT](#) [4 Building Software with GitHub Copilot](#) [5 Managing Data with GitHub Copilot and Copilot Chat](#) [6 Testing, Assessing, and Explaining with Large Language Models](#) [7 Coding Infrastructure and Managing Deployments](#)

AI-Powered Developer

Build great software with ChatGPT and Copilot

Nathan B. Crocker



MANNING