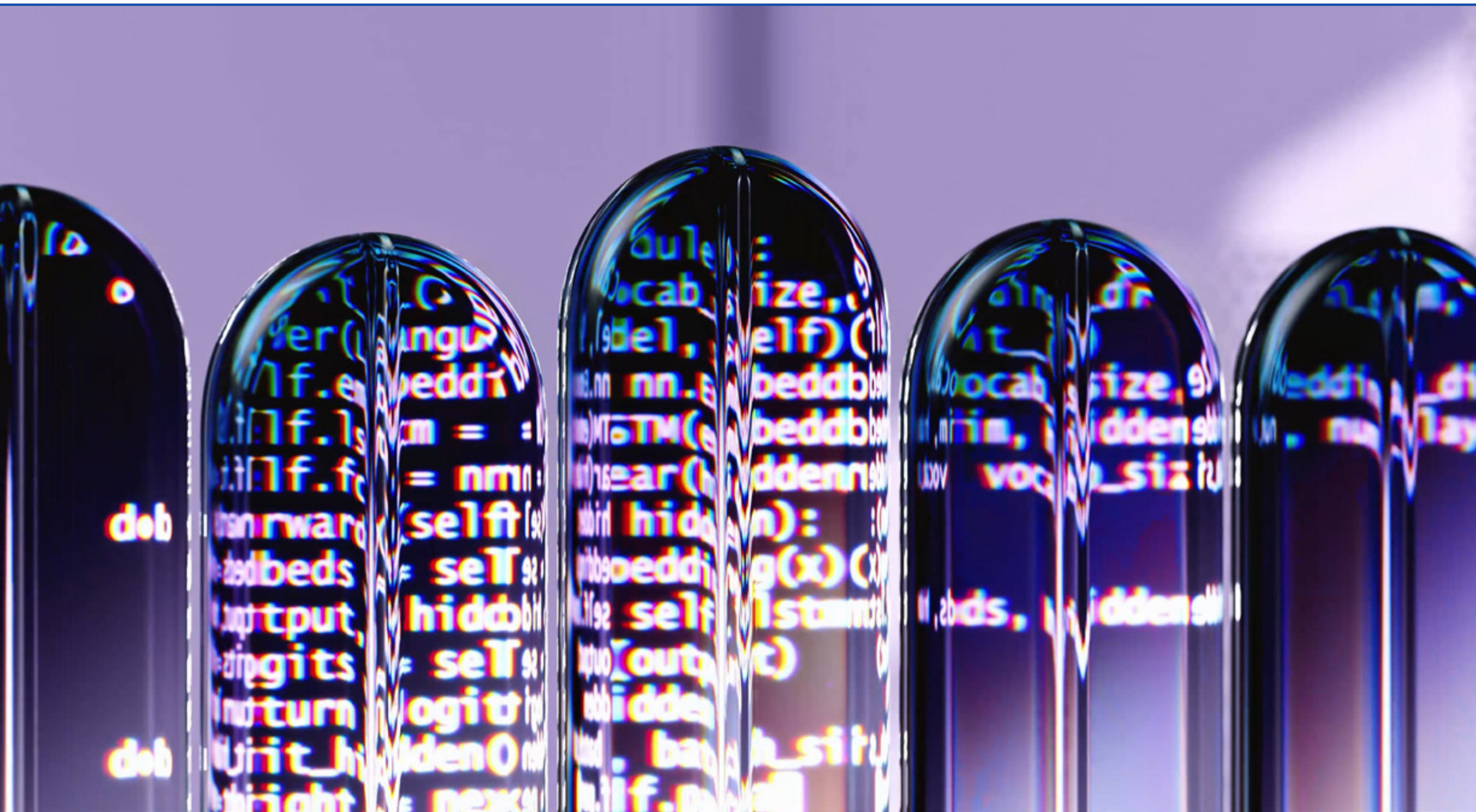
April 2024

Secure AI Framework

Securing the AI Software Supply Chain

Authors: Shamik Chaudhuri, Kingshuk Dasgupta, Isaac Hepworth, Michael Le, Mark Lodato, Mihai Maruseac, Sarah Meiklejohn, Tehila Minkus, Kara Olive



Abstract

As AI-powered features gain traction in software applications, we see many of the same problems we've faced with traditional software—but at an accelerated pace. The threat landscape continues to expand as AI is further integrated into everyday products, so we can expect more attacks. Given the expense of building models, there is a clear need for supply chain solutions.

This paper explains our approach to securing our AI supply chain using provenance information and provides guidance for other organizations. Although there are differences between traditional and AI development processes and risks, we can build on our work over the past decade using Binary Authorization for Borg (BAB), Supply-chain Levels for Software Artifacts (SLSA), and next-generation cryptographic signing solutions via Sigstore, and adapt these to the AI supply chain without reinventing the wheel.

Depending on internal processes and platforms, each organization's approach to AI supply chain security will look different, but the focus should be on areas where it can be improved in a relatively short time.

Readers should note that the first part of this paper provides a broad overview of "Development lifecycles for traditional and AI software." Then we delve specifically into AI supply chain risks, and explain our approach to securing our AI supply chain using provenance information. More advanced practitioners may prefer to go directly to the sections on "AI supply chain risks," "Controls for AI supply chain security," or even the "Guidance for practitioners" section at the end of the paper, which can be adapted to the needs of any organization.

Contents

Contents

In 2023 and early 2024, several AI models were found to be malicious—dangerous code masquerading as safe, freely shared models. The unsuspecting users who downloaded them to build AI capabilities instead received programs that harbored harmful functions, including the ability to exfiltrate data or install backdoors that would allow attackers to execute code on the users' machines.

Hugging Face, the open-source and open science platform, is addressing these attacks by pairing with security researchers to identify and fix these issues. The platform also offers a solution to protect against them by offering developers who upload models to the platform the ability to sign their models with GPG keys, a form of public key cryptography that allows users to verify the models at download time to be sure they come unaltered from trusted creators. Unfortunately, this solution isn't used often, likely because GPG signing introduces toil in the form of ongoing key management—which can be effortful and accident-prone—and also slows down the upload process.

This type of attack is not new to anyone involved in the software supply chain space. As AI expands to become a more dominant form of development, we're seeing that many of the same problems that have played out in the past for traditional software are now happening in AI—but at an accelerated pace.

We can expect to see more attacks like this one in the future as AI makes its way further into everyday products. There's good news, though. First, there are existing software security measures that can and should be applied to AI ecosystems. Second, we've learned a lot about the most useful ways to extend these solutions. As the GPG key example shows, security measures don't work if developers won't or can't use them.

In the past several years, the software industry has come together with national governments to fix security gaps in traditional software supply chains.

Often, this has meant changing common practices used by developers around the world and retrofitting existing infrastructures to harden them against vulnerabilities discovered only after they were exploited. Some of these lessons were hard-won, but thankfully they're also transferable. We have the unique opportunity now, as AI development becomes more common, to build these solutions into AI's budding infrastructure from the start, rather than address the problems later when they're harder to solve.

This white paper is one of a series describing our approaches to implementing Google's Secure AI Framework (SAIF). The paper is meant for a broad technical audience and is intended to help both AI practitioners who want to learn more about security, and security practitioners who want to learn more about AI-specific needs. We've included introductory material for both fields of practice, so experts may choose to skip the background section covering their field.

We explain our approach to securing our AI supply chain and provide guidance for other organizations to do the same.[1] In particular, we argue that AI ecosystems can take advantage of a traditional supply chain governance technique known as provenance, a metadata document to capture and secure information about what went into an artifact and how it was created.

Security-minded users can protect themselves against attacks like the one described at the start of this paper by verifying the identity of the model producer, to confirm that the model is coming unaltered from the producer they expect and trust. We believe that tamper-proof provenance is necessary for AI artifacts and data to secure AI supply chains. Provenance can also provide the auditability foundations to solve pressing concerns, such as allowing training pipelines to reason about copyright to avoid potential infringement issues. More broadly, provenance can support essential horizontals such as governance and assurance, compliance, and incident response.

[1] This paper does not cover the important topics of confidentiality, privacy, or the quality of a model's behavior, which will be covered in dedicated future white papers.

01

Development lifecycles for traditional and AI software

The following sections introduce what we mean by the software supply chain in the context of development lifecycles for traditional software (referring to non-AI software) and for AI-specific development lifecycles.

Traditional software supply chains

Many aspects of our lives and work are powered or assisted by software applications. But where does that software come from? How do we know whether we can trust it to behave as we expect? And have all of its components been acquired properly? These are some of the questions that the field of software supply chain security aims to address.

When we talk about software supply chains, we're referring to the sequence of steps resulting in the creation of a software artifact.
In a traditional software development lifecycle, a developer contributes code to a repository. Then, using external dependencies, the developer builds an executable, which is then deployed to a package repository. Later, some other developer will download the package to deploy in a production service.

Traditional software development lifecycle

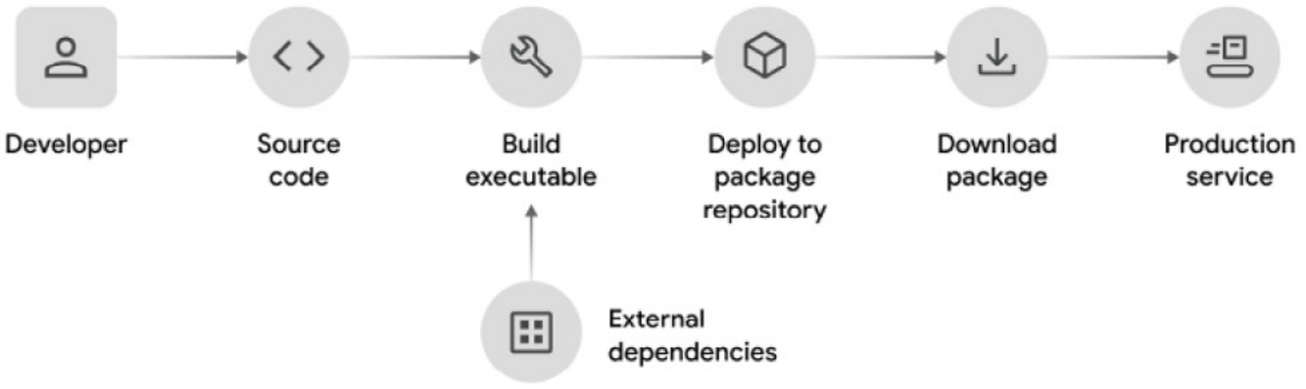


Figure 1: Traditional software development lifecycle: a developer contributes code to a code repository. Then, using external dependencies, the developer builds an executable which is then deployed to a package repository. Later, some other developer will download the package to deploy in a production service.

Figure 1 shows the process that transforms source code contributed by a developer into an artifact, through the process of a build. When we talk about an **artifact**, we mean a serialized set of bits that can be used as inputs or outputs for software—a dataset or code library, a software package or an OCI container image, a mobile app, or an AI model. When we transform input artifacts, plus some number of parameters (such as command line arguments or configuration files) into some number of output artifacts, this constitutes a **build**.[2]

If not secured, nearly each stage in this process can be the source of a supply chain security problem. Russ Cox, a Google Distinguished Software Engineer and author of the Go programming language, frames software supply chain security as being concerned with hardening a supply chain against two classes of problems: attacks and vulnerabilities.

How are attacks different from vulnerabilities? In the context of supply chains, the term **attack** indicates nefarious alteration of software *before* it's been delivered.

For example, if an engineer at a software company surreptitiously replaces a new OS version with a backdoored image, and the bad image then gets shipped to customers, we would consider that an attack.

The 2020 SolarWinds hack was a notable example of a software supply chain attack. Intruders installed a Trojan horse in the software update process for critical network software, leading to network backdoors across thousands of companies and the US government.

In contrast, **vulnerabilities** are often unintentional flaws in design or implementation of software or its dependencies, which may become visible to humans only after the software has been shipped.

[2] NIST SP 800-204D, "Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines" provides a good generalized schematic model of how software supply chains comprise individual transformation "steps." See https://csrc.nist.gov/pubs/sp/800/204/d/final.

We think of these as exploitable weaknesses in software that accrue due to dependencies of that software—a problem that compounds, since many projects have a large number of dependencies, which themselves have other dependencies of their own, and so on.

For example, Log4Shell was a supply chain vulnerability that affected millions of Java-based applications and devices. This vulnerability sent the software industry scrambling to patch and update affected Log4j packages when it was discovered that a feature set created for innocent reasons could be exploited to gain remote code execution on a remote host merely by entering some trivial inputs on a web form. This vulnerability was in part so impactful because Log4j was a package that often occurred deep in a piece of software's dependency graph, as many as twelve layers of dependencies down.

There are two main areas of concern in supply chain security: dependency tracking to enable fast reaction in case of compromise, and tampering protection to prevent compromises through malicious modifications to software artifacts.

Dependency tracking

Dependency tracking is useful for managing both vulnerabilities and software licenses. When a software project imports a pre-existing software library to enable parts of its functionality, it may inherit security vulnerabilities as well as licensing restrictions from this dependency.

If a widely-used software library or component contains unexpected behavior that can be exploited, then any other packages which rely on it *might* be vulnerable. However, this isn't guaranteed—for example, the vulnerable code path might not be exercised in many of the packages which include it.

Figuring out if vulnerable code is being executed with certainty is hard, as it is often an undecidable problem. Approximations rely on good instrumentation and logging practices in order to enable post-hoc analysis. Additionally, even when packages have been determined as definitely vulnerable, patching is a difficult process: the primary bug needs to be fixed, and then the patch needs to be rolled out gradually and iteratively to all the downstream dependencies which are implicated. As the Log4Shell incident showed, this is a difficult process when there are many layers of dependencies between your software and the affected package, since some language ecosystems require multiple intermediate updates before a downstream package can be fully patched.

The Software Bill of Materials framework (SBOM) helps us encode dependency relationships by providing a list of "ingredients" in a piece of software.

SBOM isn't a complete solution yet, but it enables further progress on industry-wide dependency discovery so that we can track, measure, and eventually remediate the spread of vulnerabilities across our software ecosystems.

SBOMs can also be used to track software licenses associated with a software artifact. Software licenses are used to specify the intended use of open source libraries in applications that include them. When an organization tracks all of the software licenses used in its software artifacts, and takes care to use the software appropriately, it can ensure that it respects the intentions of open source library authors.

Tampering

When a user downloads a software application, how can they ascertain that it behaves according to its intended source code specification, rather than including nefarious changes?

Figure 2 demonstrates some of the critical points at which an attacker or a malicious insider might inject unexpected code into an application throughout the software development lifecycle.

Supply chain risks in traditional software

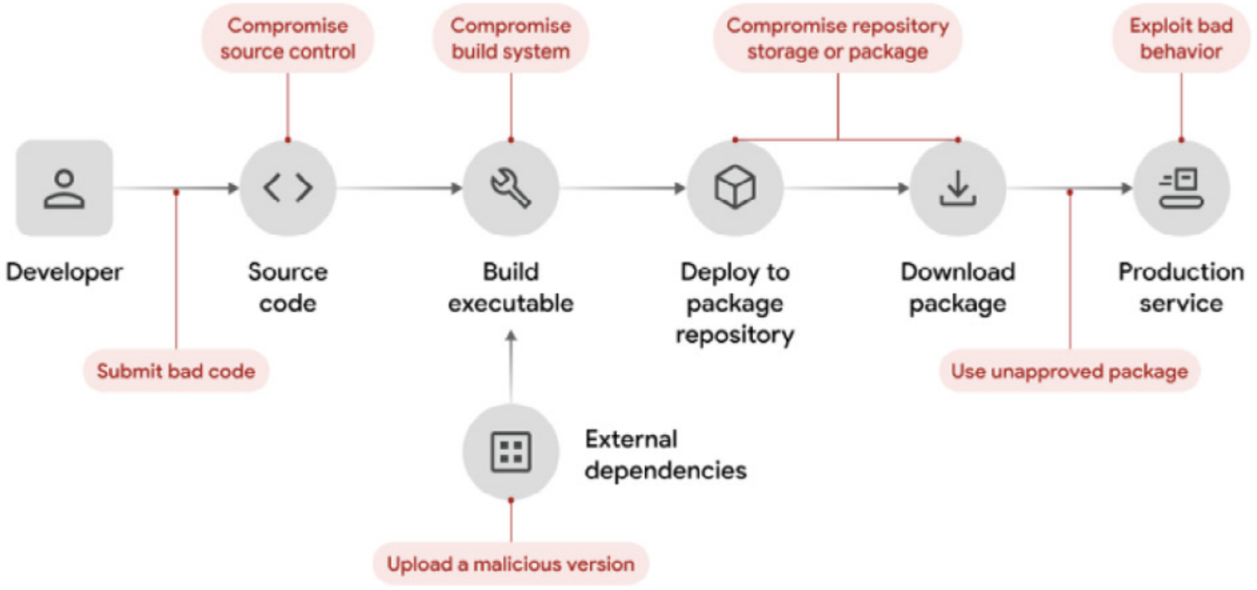


Figure 2: Supply chain risks associated with creating a software artifact.

In particular, an attacker could modify a software artifact's behavior by making unapproved changes to the code, the build system, any relevant binary dependencies, or the package repository in which the artifact is ultimately published. Solutions like Supply-chain Levels for Software Artifacts (SLSA) help software providers adopt best practices in maintaining a chain of cryptographic custody linking source code to the eventual software applications. This chain is represented by a **signed provenance** document: a tamper-proof attestation of how a software artifact has been produced.

Signed provenance is designed to supplement SBOMs by imposing additional tamper-resistance mechanisms on the chain.[3] Provenance also supplements the information about the source of the artifact and required runtime dependencies with further information about the build (the tools and processes used to create the artifact) and build dependencies. In short, an SBOM provides information about *what* is in an artifact, whereas signed provenance provides tamper-proof information about both the *what* and the *how* for details around an artifact's creation.

[3] https://slsa.dev/blog/2022/05/slsa-sbom

AI software supply chains

The following section applies the analysis of traditional software supply chains to AI software supply chains, to illustrate the similarities and differences.

These two domains of concern discussed immediately prior—dependency tracking and tampering—also apply to AI software supply chains, but with slightly different framing for AI-specific aspects:

Dependency tracking: As with traditional supply chains, it's important to find and fix bugs that get introduced into AI artifacts and infrastructure. With AI, though, a new class of dependencies emerges: the datasets which have been used to train a model. The tracking ability is relevant not just for security and privacy concerns, but also for governance of use restrictions based on copyright, similar to concerns related to software licensing.

Artifact integrity: The ability to sign and verify an AI model and know that it hasn't been tampered with, similar to signing a traditional software artifact.

To understand how these concerns differ when applied to AI, let's take a look at how AI applications are usually developed, without going into particularities about specific models.

The central concept for AI-powered applications is the **model**[4]. From a high-level view, a model can be viewed as a pairing of code and weights, created as part of a **training process**, that is only useful when deployed in AI-powered applications. The purpose of a model is to extract statistical patterns from data and use these to make predictions (also known as **inferences**) on new data for applications that use AI. At this high level, the process looks like the following diagram:

ML model development lifecycle

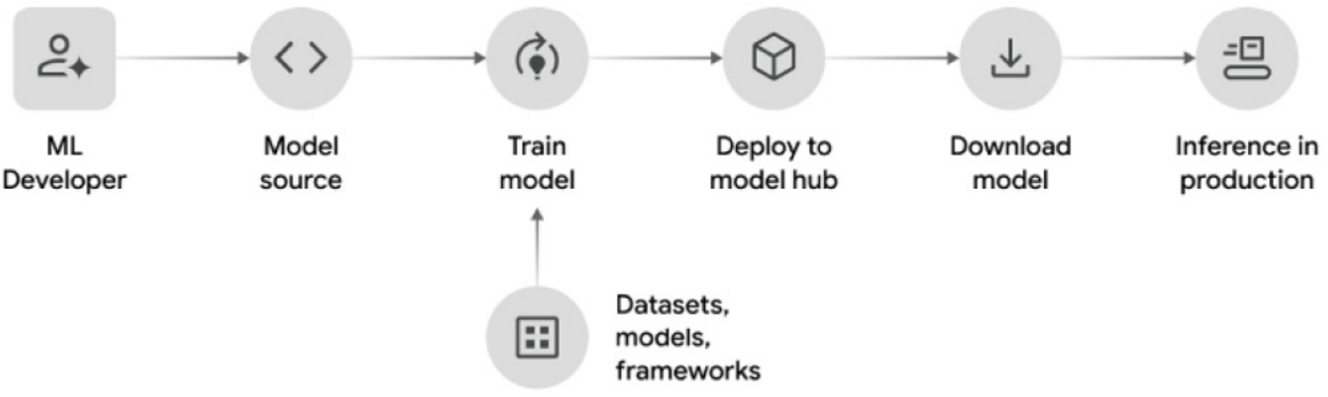


Figure 3: The lifecycle of an AI model: an AI developer chooses a model architecture and uses external datasets, pre-trained models, and a model training framework to train a model. The model is deployed to a model hub from where it will be downloaded later to be used in production for inference.

[4] It should be noted that securing the model alone is not enough to secure an AI system. You also need to protect its training data, the infrastructure used to train and serve the model, and the application that uses the model.

The following brief introduction to training AI models focuses on aspects of development that are relevant to supply chain risks. There are multiple types of models, of various formats, sizes, and different purposes; however, the main supply chain concerns apply similarly for all cases. As the diagram indicates, the high-level story is quite similar to the development of traditional software.

Datasets

To begin training a model, we need data from which to extract patterns. *Datasets are arguably more important than the model architecture source code*: no matter how complex a model is, it can only perform well when trained on suitable data. Thus, **data sourcing** requires practitioners to ask a few questions early on:

- What is the intended use case for the system—what task does it help with?
- What questions need to be answered for the task the model needs to perform?
- What data could train the model to answer these questions?
- What sources of data might fit the needs of practitioners and end users?
- Is the data high-quality, complete, accurate, and relevant?
- Are there any ethical and legal issues associated with the datasets used in training?

Once adequate data sources are identified and acquired, practitioners ingest them into local storage to enable faster training. Here, we see the first supply chain risk: (a)[5] data could be maliciously poisoned before or during its ingestion process.

Once the data is ingested, it usually needs cleaning and transforming, processes that are collectively known as data augmentation. The data might need new labels to help train the model; it may contain low-quality, duplicated, or inconsistent records; or it might be in a different format than the task requires. If a practitioner is using multiple datasets, they might also need to resolve inconsistencies in formatting and content.

[5] The risks discussed in this section are labeled with the same letter in Figure 5.

Human and algorithmic labelers can be used to label, filter, or transform the data. From a supply chain perspective, humans might maliciously mislabel the data. Alternatively, an algorithmic labeler might have a bug that results in improper data transformations. All of these then impact the performance of the trained model. This is another supply chain risk: (b) unexpected (malicious or incorrect) training data can be used to train a model. From the point of view of the training process, this risk encompasses data poisoning (risk (a) above), but this one is more general, as it impacts an artifact seen by end-users.

Some examples of transformations include:

- Removing duplicate records
- Supplementing missing fields from another dataset
- Changing the format or scale for specific fields
- Adding new examples by transforming existing data points (e.g., rotating an image)
- Increasing the amount of data available for training by generating synthetic data using a different model

Since all of these transformations result in datasets that are different from the ingested versions, from a supply chain integrity perspective, we need to keep track of these operations. This gives rise to the notion of **lineage**: metadata to capture all pre-training transformations that have been performed on datasets and their resulting models. Lineage resembles provenance in the traditional software supply chain, but provenance is broader, since it also covers infrastructure metadata and cryptographic signatures for inputs and outputs. Since data used during training is critical to a model's post-training performance, it is essential that we capture all the supply chain lineage and provenance information related to dataset operations. Lineage and provenance will form the foundation for governance of AI models, including establishing policies and controls for what copyrighted materials are acceptable for training a model.

Models

Models are the central concept in developing AI-powered applications. At a very high level, a model is a collection of **weights**—parameters that determine how each **feature** (data attribute, data column, etc.) influences the output. To train a model means adjusting the weights until the predicted output is close to example **target labels** from the training dataset. The distance between the example labels and actual labels is measured by a **loss** value, which the training process aims to minimize. (For an introductory explanation of the process, see this Machine Learning crash course.)

As a light-hearted example, imagine the task of selecting a daily lunch for a picky eater. You might consider many factors: what they ate last; the suggested food's temperature; its color; the flavors it contains; nutritional value; texture; the time of year; whether the picky eater is dining alone or with friends. Some features will play a larger role than others, or sometimes their weights might depend on other feature values.

By categorizing all these features and tracking the picky eater's preferred menus, you can learn the features' weights over time, allowing you to gradually infer a decision tree that helps you plan better menus in the future.

The same concepts of features, target labels, and loss are also used when developing more complex models, including large language models (LLMs) and multi-modal foundation models (models trained on large amounts of data to perform a large variety of tasks). Here, the model performs multiple computations, merging data and weights to create intermediate computations. The results of these computations are then mixed with other weights and computations in an iterative process until the final prediction can be produced.

The resulting model architecture is called a **computation graph**: it represents the forward flow of data from input through to prediction. This graph records all the computations that occur during inference.

Training large models from scratch is expensive, taking massive amounts of time and resources. Frequently, developers will start with a pretrained model to reduce this burden and then construct a new model on top. For example, a developer could take a pretrained model and perform transfer learning on it.

Transfer learning teaches a model trained for a specific task to perform a different task. **Finetuning** is a type of transfer learning that freezes most of the model weights and updates only the last few computations in the model architecture.

Transfer learning process

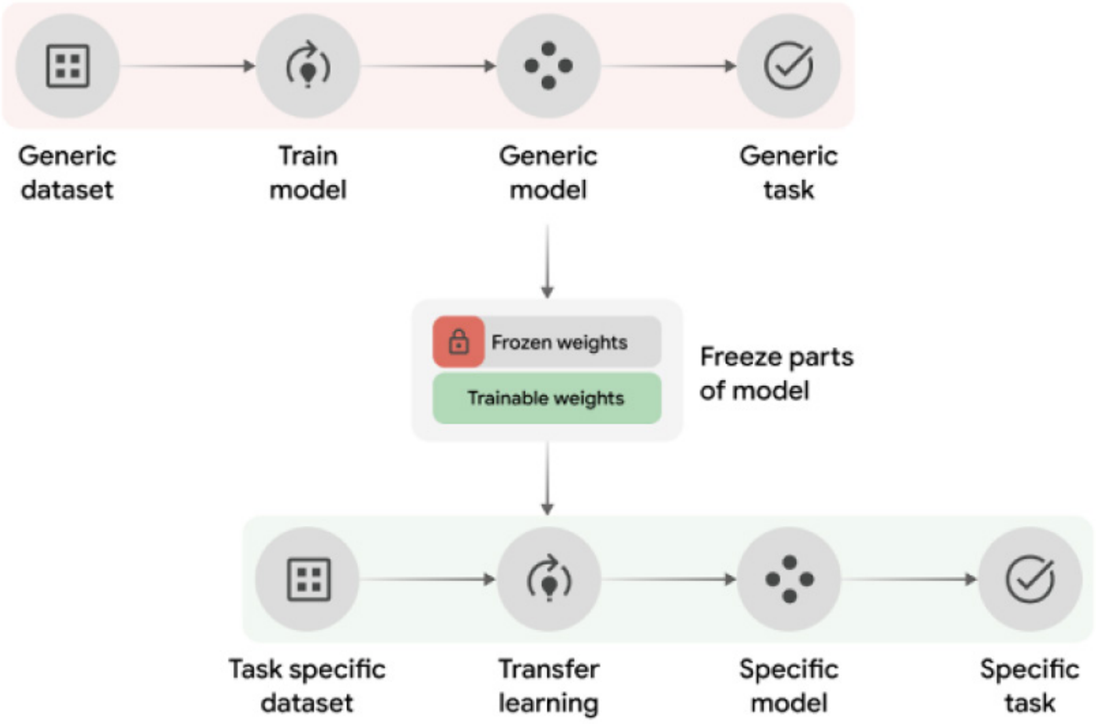


Figure 4: An overview of the transfer learning process. A generic model is trained on a generic dataset to perform efficiently on a generic task. For finetuning, the first stages of the model weights are frozen and the model is trained again on a task-specific dataset.

These techniques allow models to be taught new tasks with less cost than training from scratch. For example, a developer could take a model that was trained against a large general-purpose image-recognition dataset and finetune it against a much smaller set of medical diagnostic scans to yield a model that detects abnormalities in X-ray imaging. Similarly, the components used for language and grammar representation in a LLM can be reused in a new model that performs well on specific categories of text generation.

As we finetune pretrained models for specific applications, we need to make sure that the supply chain metadata is properly recorded. This means capturing provenance for every training dataset, metadata from any training processes, and (because a generic model might have been trained by a separate entity) the provenance of the pretrained model. A generic model that is not well protected or provenanced could represent a supply chain threat for any models derived from it—(c) it could have been maliciously trained (to include backdoors or perform poorly on specific tasks), or it could have been tampered with between training and finetuning.

Other model development scenarios involve combining the output of multiple models into a larger model. As an example, consider a technique called Mixture of Experts (MoE). With this technique, each individual model can solve a part of the problem space, but does not perform well in other cases (such as when specific types of features are present). By synthesizing the predictions of each individual model, we can achieve a model that has a better performance overall. When using MoE, we again need the provenance metadata for all involved models in order to have full visibility into the supply chain of the final model used in production.

Model serialization

Once the model is trained, we put it to work by adding it to a production pipeline. For example, we could create a web application to generate real estate listings based on some key inputs. In order to create such applications, we need to serialize our model and may then choose to store it in a hub, from which it can be downloaded for each new application.

Serialization allows us to transfer models into new environments, which is useful because the hardware and infrastructure used for training is generally different from production inference infrastructure. From the supply chain integrity perspective, we need to ensure that the model cannot be tampered with while in storage (risk (d) in Figure 5).

One approach to model storage is to record the weights in a file, which each application parses before using the model. This is known as **checkpointing** and is widely used during training, where AI practitioners periodically save the weights from long training loops. If the process aborts or misbehaves, training can be restarted from the last set of weights or **checkpoints**.

There are a few approaches for serializing models into checkpoint files:

- Using predefined serialization features provided by a language—for example using `pickle` in Python. Depending on the implementation, this can be insecure so we don't recommend it—the SafeTensors library is a good replacement.
- Using a library to store the weights in a format that is understandable by the library—for example, for a model trained using `numpy`, we could save the checkpoints in a file following the NPY format—which can be insecure in some cases too. However, this means that every inference application must reuse the same code used during training. Because this creates a hard dependency between the training code and the inference code, we don't recommend this for more advanced models.
- Packaging both the weights and the model architecture into a single entity. This can be a single file (for example a flatbuffers file for `.tflite` models used by TFLite or a zipped package for `.pth` Pytorch models) or a structured collection of files and directories (for example, a TensorFlow SavedModel). Sometimes, large models that would otherwise be stored in a single file are also split into multiple files to speed up loading. Note that even these formats can be insecure, for example `.pth` files can use `pickle` and `SavedModels` can use `Lambda` layers to run arbitrary code.

Bundling weights and architecture into a single package format allows developers to transfer models between applications more easily, as long as they integrate an interpreter for the model format in their AI-powered application. The interpreter parses both the model structure and the weights to construct the appropriate memory layout to perform the inference.

A single package also allows the interpreter and the framework used to train the model to evolve separately. As long as compatibility is maintained, models trained with one version of the framework can be used with an interpreter matching another version. For some of the existing serialization formats, it's possible to achieve both forward and backward compatibility, achieving a full decoupling between the training process and the applications that use the models in production.

Since model serialization represents creating a new artifact in our supply chain—namely a checkpoint—this is another place where we should record provenance memorializing the operations performed (risk (c) in Figure 5).

Recording complete provenance for each new artifact or checkpoint helps developers track risks introduced during storage or model serialization.

After a model has been recorded, it might also undergo **model quantization.** This process takes a fully trained model and shrinks it by converting its weights into low precision integers, and, in some cases, also replaces operations in the model's computational graph with operations that can operate on the quantized weight. This increases the efficiency of the model, especially when it gets deployed to embedded/mobile applications, by allowing integer operations to consume fewer computational resources and less time. The loss of precision from this conversion is not significant enough to cause the model to mispredict during inference.

The process of quantizing a model is also building a new artifact, so we need to record the associated supply chain metadata to maintain the provenance of the updated model (this is also represented in Figure 5 by risk (c)).

Overall, we need to remember that *models are not easily inspectable:* the behavior of a model is heavily influenced by its weights, yet given the large number of weights and binary format, it is not humanly possible to analyze the weights to predict what a model may do. In some storage formats, it is also difficult to analyze the computational graph. Instead, we adopt the point of view that *models are programs*; they are similar to bytecode that is interpreted at runtime to produce some valuable set of behavior. However, unlike traditional software where it is feasible to understand a binary via reverse engineering, models are opaque, meaning their behavior can be only partially understood by observing a small fraction of all possible inputs. Given the expense of building models, there is a clear need for complete supply chain provenance information. In case of an attack on the training platform, for example, we can use this information to quickly identify which models need to be analyzed and retrained to remove potential tampering (risk (e) in Figure 5).

Training framework

AI practitioners generally don't write code from scratch to train models. Instead they use libraries optimized for operations that can take advantage of the available hardware. Powerful libraries—such as JAX, TensorFlow, and PyTorch—are able to use hardware accelerators—GPUs and TPUs—on the training host machine to significantly improve training speed. For large models, the framework can distribute computation across multiple hosts, managing scheduling and network communication optimally.

These features make training frameworks complicated, allowing opportunities for vulnerabilities to be introduced (risk (f) in Figure 5). The training libraries are therefore a critical part of the supply chain, since they can produce models that have been affected by a vulnerability.

Recording provenance about training frameworks allows us to identify models trained with frameworks later found to have a bug in the implementation of some math operation, or in the way they compile model-specific code to run on hardware accelerators.

Evaluation

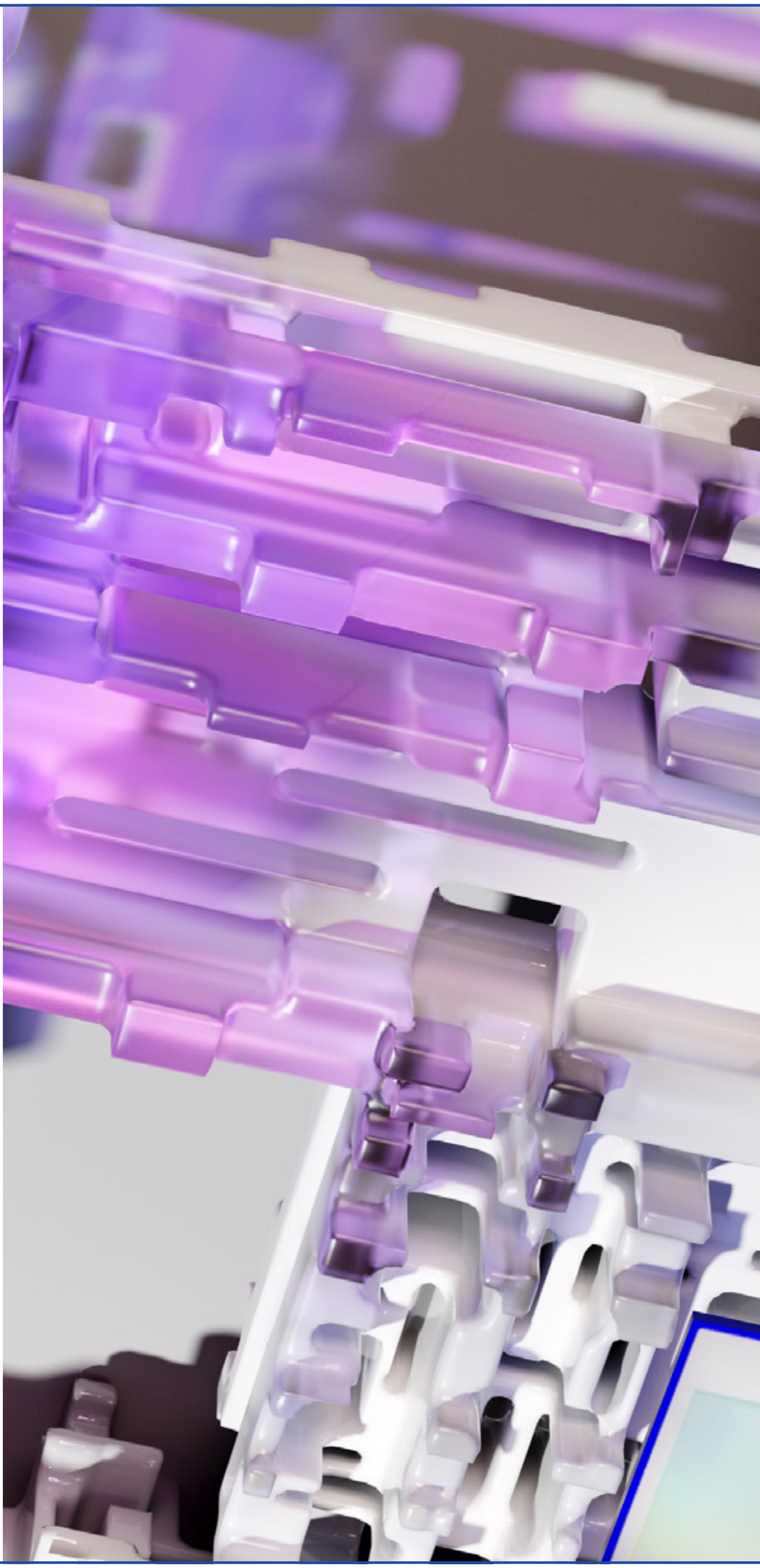
When training models, we want to make sure they perform effectively on examples of their task, without overfitting[6] to the specific examples they were trained against. There are multiple junctures during the process where we evaluate performance:

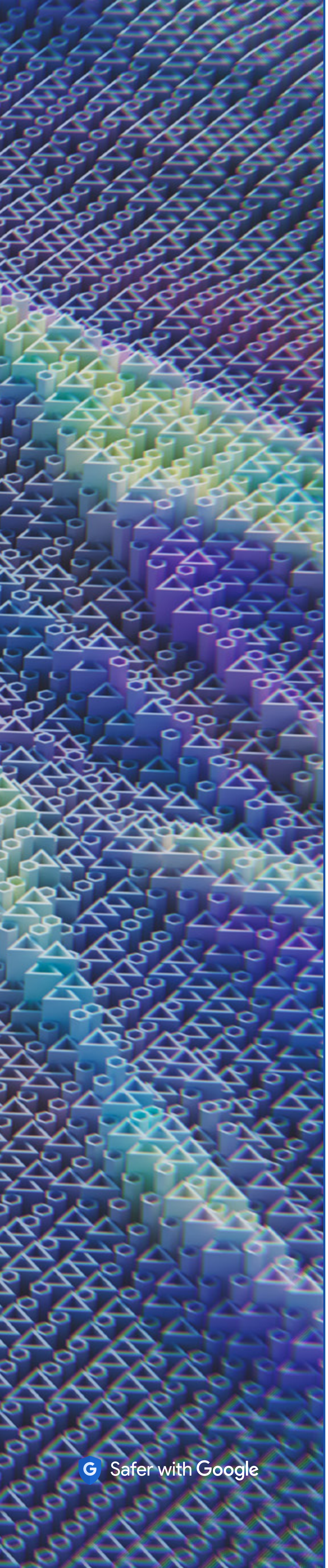
- **Automated testing during training:** This uses a dedicated portion of the training data. Instead of giving that data to the model during the training loops, it is used to evaluate the model at every checkpoint, as a proxy for the model's performance on data not seen during training.
- **Human evaluation during training:** LLMs and foundation models are also evaluated by humans during training using reinforcement learning with human feedback (RLHF). Periodically, a checkpoint is used to answer a variety of prompts, and humans rate the answers. This provides a signal to the model on what answers are suitable for the prompt, and the training process will update model weights accordingly.

Since RLHF and similar techniques influence the resulting model, we should capture provenance to incorporate information about these processes in the AI supply chain.
If test data were manipulated, or a human rater maliciously encouraged incorrect answers, this could influence the behavior of the model (risks (a), (b), (c) in figure 5); recording provenance lets us detect the impact of such manipulations when they are discovered.

[6] Overfitting is when a model performs exceptionally well on the training data but fails on new data.

Larger models, such as LLMs and foundational models, are also evaluated after release, often by third parties. These evaluations are similar to integration tests or acceptance testing in traditional software: they don't change the software, but they allow an organization to decide whether it performs as expected before admitting or deploying it. Organizations with an emphasis on production hygiene and observability may choose to perform such tests in a trusted execution environment and record the results in a signed attestation, so that they can ensure models have been adequately evaluated before use (risks (g) and (h) in Figure 5, where (h) occurs once the model has been used inadequately in production).





02

AI supply chain risks

This section discusses in more detail how the AI development lifecycle described previously can lead to AI-specific supply chain risks.

In many ways, the process by which one trains, publishes, and serves a model strongly resembles the traditional **software development lifecycle** (SDLC). We can consider that the training process (or data transformation when doing dataset enrichment) represents a "build." The "sources" and "dependencies" of the build are represented by the training framework, the code used to define the model architecture, and the datasets. Finally, the resulting "package" is the model (or training data in the case of dataset augmentation).[7]

[7] Again, the "SSC Model" in §2.4 of NIST SP 800-204D, "Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines" is a useful abstraction of these concepts — with equal applicability to AI as to traditional software.

Similarities to traditional supply chain risks

In the previous section, we have shown that training a foundation model involves multiple components and processes:

1. Starting with datasets, we perform multiple data cleaning and data augmentation steps.
2. We choose a framework with which to train a new model, combining the cleaned and augmented data with previously trained models.
3. The model is recorded as a checkpoint, and possibly quantized into a smaller footprint.
4. The checkpoint is stored in a model hub.
5. The checkpoint then serves future training steps or gets deployed in AI powered applications.

Every step can be affected by unintentional flaws or design choices that can result in supply chain compromises. To summarize the risks we identified as we walked through the model's lifecycle previously:

- Data could be maliciously or inadvertently poisoned, either at ingestion or during curation and cleaning.
- The training platform might be vulnerable to attacks.
- Training frameworks and libraries may contain vulnerabilities or backdoors which affect their computations. For example, model checkpointing or quantization code could introduce changes to the model architecture or weights in possibly security-sensitive ways.
- Human raters, or automated testing steps during training, could introduce buggy or malicious inferences.
- Model hubs could be compromised, allowing a malicious developer to poison model weights or datasets, thus affecting both production uses as well as future training steps using pretrained models.
- When deploying the model to create AI powered applications, developers might inadvertently use inadequately trained or evaluated models if they receive a different model than expected.

Any of these risks could make a model in production vulnerable to exploits.

Supply chain risks in foundation model training

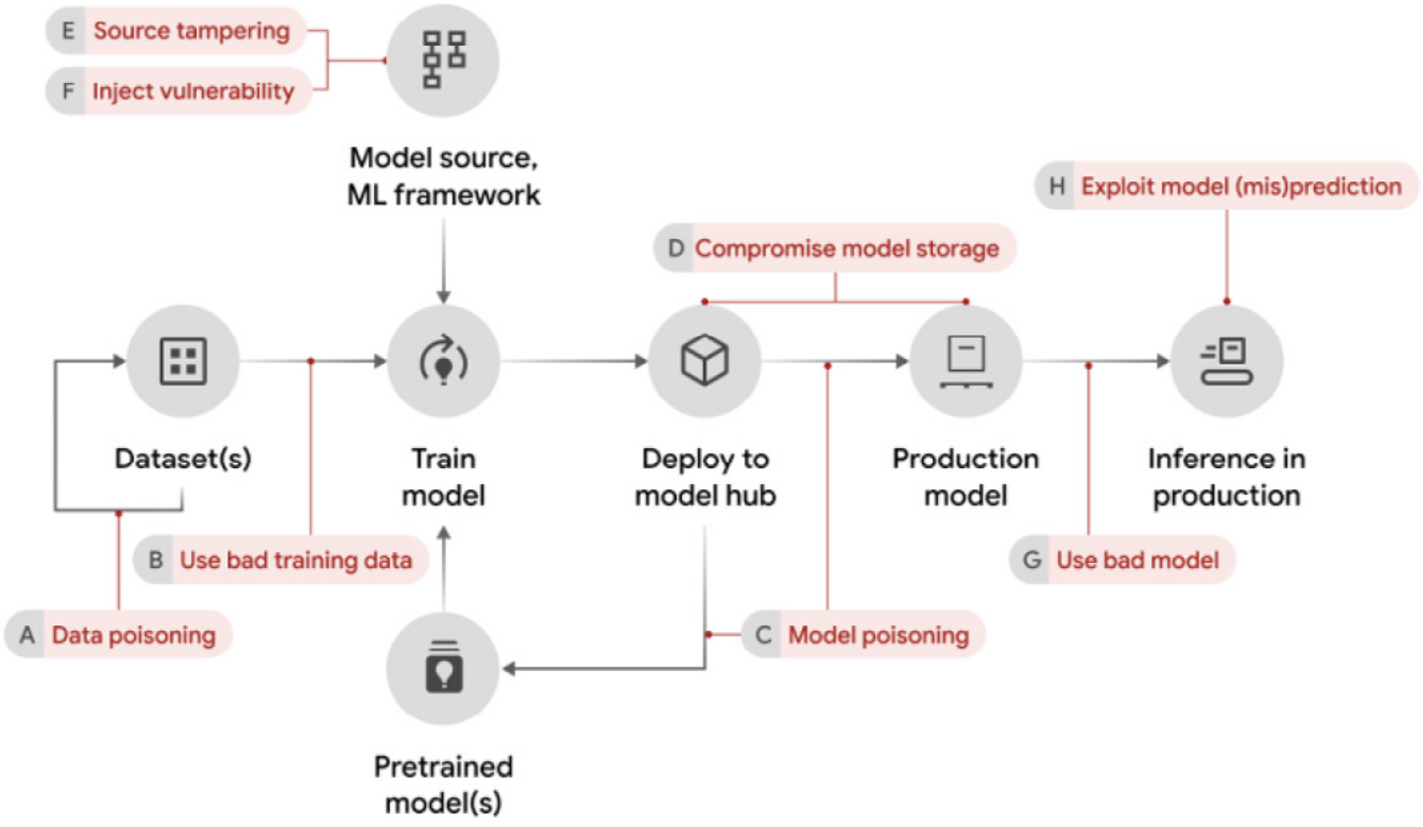


Figure 5: The overall training of a foundational model and the associated supply chain risks

In order to secure this complex system and ensure security for model supply chains, we should analyze how every step of this process works as a whole (Figure 5) and individually. Since training is the common factor in any AI supply chain, let's look at the training supply chain diagram and the associated risks as representative of the risks across the entire model development process. This diagram of risks expands on the AI model development lifecycle shown in Figure 3:

Supply chain risks in ML

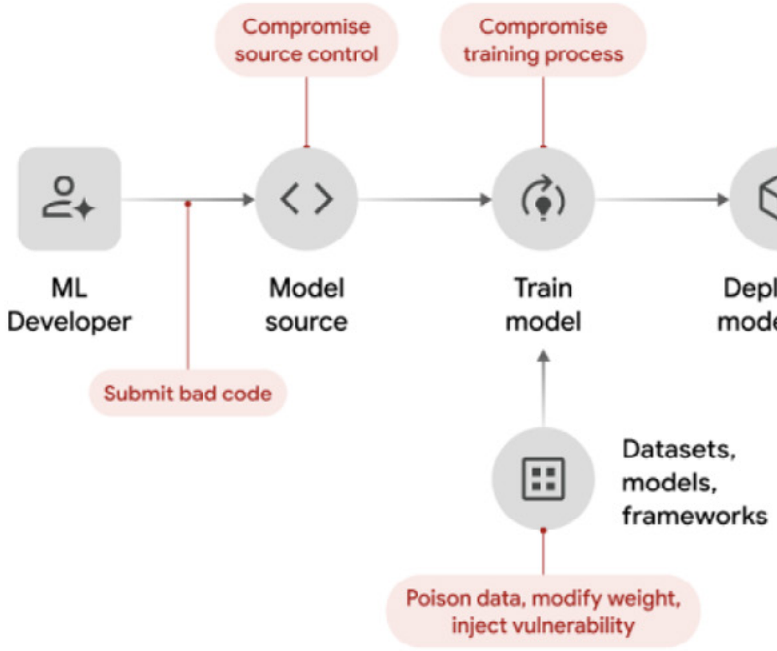

Compromise source control
Compromise training process
Compromise model storage
Exploit model (mis)prediction
ML Developer
Model source
Train model
Deploy to model hub
Download model
Inference in production
Submit bad code
Use bad model
Datasets, models, frameworks
Poison data, modify weight, inject vulnerability

Figure 6: Supply chain risks associated with training a model

When we compare the AI supply chain in Figure 6 with the risks from the traditional software supply chain from Figure 2, we notice clear parallels: namely, they both incorporate notions of code, version control, deployment to a repository or hub, and eventual download by a package consumer. These similarities inform the risks and controls discussed later in this paper.

Differences specific to AI development

As we just explained, AI development shares a common shape with traditional software development. However, there are also some practices in the software development life cycle which diverge from the current state of AI development.

Let's take a look at some characteristics of traditional software development that may not directly transfer to AI:

- **Source control:** Code is usually stored in a version control system. This lets developers collaborate on a shared codebase, track the state of the codebase over time, and roll changes back or forward as needed.
- **Code review:** Many organizations use code review as a tool to ensure that new code changes match their standards for readability or correctness.
- **Hosted, scripted builds:** To transform the code into a package, many organizations use a hosted build system that follows a set of predefined configuration steps. This automation allows for higher consistency across developers and release cycles, and it also reduces the burden of maintenance on each individual developer.
- **Short, cheap build cycles:** If a developer discovers that a recently-built package contains bugs, or decides that they'd like to change some functionality, they can start a new build without incurring much additional overhead (perennial complaints about build latency notwithstanding).

When we look at AI development, these attributes may differ in significant ways:

- **Immature version control for datasets:** The ecosystem for storing, changing, and retrieving datasets is generally less opinionated and robust than the corresponding ecosystem for code management. The huge size of most datasets also adds additional constraints to the robustness of version control solutions. This means that the contents of a given dataset may change through cleaning, copying, and extension, often without incurring an explicit semantic versioning bump.
- **Human review for data is challenging:** Training datasets are large and may be updated frequently. It could be expensive or infeasible to perform human review with high confidence after every data transformation.
- **ML training is not usually fully scripted:** ML training often comprises a series of ad-hoc incremental steps that are not recorded in any central configuration. It may also span a series of systems and frameworks, rather than running entirely within the confines of a single hosted build system. Training can also rely on specialized hardware, which makes it more difficult to apply sandboxing techniques commonly used in hosted build systems. The gold standard of hermetic, reproducible, and deterministic builds is much more difficult to achieve when training ML models.
- **Long, expensive training cycles:** Because training is expensive, many training libraries support additional ways to “sideload” new data or code into already-running processes. This means the control plane for injecting inputs into ML training is more diverse and complicated than a traditional build system.

Since data version control is less mature than code source control, it’s harder to track the provenance of datasets as they are prepared and made available for training. Likewise, since training processes can be more complicated than build processes, it becomes correspondingly more difficult to track the provenance of model inputs and outputs as they flow through training.

03

Controls for AI supply chain security

The following section discusses the available controls that can protect against key AI supply chain risks.

We believe that *traditional supply chain solutions can and should be extended to apply to AI development.*

Though there are differences between traditional and AI development processes and risks, we can adapt traditional approaches, such as applying protections designed for code to also cover data. For example, we can extend the concept of provenance to cover data transformations, extending on what ML practitioners call lineage. There's no need to reinvent the wheel.

An end-to-end approach to AI supply chain security should address the two needs discussed previously in this paper: tracking all dependencies of the AI-powered applications, starting from data and ending with the production model, and ensuring the integrity of all artifacts.

In order to meet these needs, an organization should work toward being able to confidently answer a series of questions about AI models used in production:

- Who created the model?
- What was the system used to train the model? What systems processed the dataset?
- Has the model been changed in any way since it was published?
- Which version of the model is this? Which versions are in production?
- What data sources were used for training, testing, and evaluating the model?
- How were the data sources processed or cleaned before training?
- Which code frameworks were used to train, test, and evaluate the model?
- Which evaluations have been run on the model?
- Are the datasets appropriate for use in the given training context?
- Do the datasets have any specific attributes (copyrighted data, licensing restrictions, location data, etc.) which require specific care and approvals before training?

Guiding principles

To answer crucial questions about supply chain security, an organization can focus on areas where they can move the needle of AI supply chain security in a relatively short time.

This includes:

Protecting integrity for the production systems[8] which process, train, or serve AI models. Organizations need to ensure that all the infrastructure used in pre-processing, training, fine-tuning, or serving is secure and resilient against software supply chain threats. Because this is a fast-moving space, it can be challenging to ensure that systems remain compliant as they evolve or add new features.

Cataloging provenance for all datasets and AI models. In addition to securing the infrastructure used by AI training and serving workflows, we also need to understand the provenance of the datasets (used either for training or benchmarks) and models themselves. What are all the inputs used to produce a model? Which datasets does the organization store and use, and what are their relevant properties?

Tracking the provenance of AI artifacts means comprehensively recording usage relationships between them. Besides enabling supply chain integrity, provenance also helps speed up launch approvals (determining if an AI-powered application can be released as a user facing product) and debug behavior that is observed in production (e.g., determining if a bug in a compiler used during training or serving a model affects the results of an inference of a model).

Protecting models against tampering and datasets against poisoning. Even if a model is trained on secure infrastructure, it can still be vulnerable to further tampering.

[8] Chapter 14: Deploying Code of Building Secure and Reliable Systems is a good reference for how to protect the integrity of production systems for traditional software. Many of the items presented there apply to the AI world, too.

For example, an attacker could copy the model and fine-tune a new version against a carefully curated set of data that causes it to issue objectionable inferences, or an insider could surgically overwrite some of the model weights to cause unexpected behavior. Targeted supply chain attacks have already been demonstrated against models in the wild, with one proof-of-concept attack that surgically modified a popular model to spread misinformation and uploaded it at a legitimate-looking repo path to deceive users into downloading it.

For models as well as some datasets,[9] it's possible to mitigate against poisoning by annotating artifacts with signatures and provenance. This also speaks to the importance of robust data cleaning processes: if the data cleaning processes are properly tracked in provenance documents, an organization can detect poisoning by insiders. An organization can also mitigate against poisoned data that is imported from external sources at the start of the supply chain via early data cleaning practices.

Discovering and patching or replacing buggy or vulnerable artifacts. Because training undergoes many iterations, adding more data and code throughout the process, vulnerabilities could creep into a model in many ways. To prevent exploitation and enable remediation, we must track all the inputs—both code and data—that were used to produce models. Likewise, we must maintain an accurate inventory of all the models that an organization uses or produces, so that we can respond to vulnerabilities as they arise.

In addition, because AI development relies on large-scale data processing, it also inherits classes of problems related to data curation and sharing.

Preventing accidental or malicious data rights infringement. This concern has become increasingly important in the AI context. In particular, we highlight the need to ensure that training processes can consume only approved datasets.

[9] Datasets can be gathered in two different ways. The first type of dataset can be imported as relatively stable, versioned artifacts: e.g. a public domain dataset that is imported, cleaned, and stored at discrete intervals. This kind of dataset can be treated similarly to other kinds of software artifacts, and lends itself to signatures and provenance tooling quite nicely. However, not all data fits this shape: for example, if you consider dynamically-generated database records, such as anonymized search logs that are continuously written based on real-world interactions, this might demand a more specialized approach.

Google’s approach: BAB, SLSA, and artifact integrity

The following section discusses Google’s internal approach to supply chain security for AI models.

This paper focuses primarily on the integrity and data provenance aspects of the AI supply chain concerns discussed earlier. Vulnerability management is an expansive topic that will be covered in more depth in a separate white paper, but the solutions discussed here support and enable a robust vulnerability management program.

Our approach to AI integrity is to extend the existing supply chain security work we’ve done over the past decade using Binary Authorization for Borg (BAB), Supply-chain Levels for Software Artifacts (SLSA), and next-generation cryptographic signing solutions such as Sigstore. We also use specific tooling to catalog data and capture the information about AI artifacts that could be used for security controls, governance, and other AI-specific concerns such as copyright infringement.

BAB and SLSA address nearly every risk discussed in this paper; the remaining risks are addressed by data cataloging for AI artifacts.

Binary Authorization for Borg

BAB, or Binary Authorization for Borg, refers to the controls that ensure that only authorized binaries can run on our production cluster management system (“Borg”).[10]

BAB was first conceived to protect against insider threats. Within development teams, everyone knows and trusts one another; yet, statistically as an organization grows, the risk also grows that a bad actor might try to tamper with code or data. In addition, there is the risk that a development team member’s workstation might be compromised by an external attacker who then uses the team member’s privileges to tamper with code or data.

[10] BAB is also known as BCID, which stands for Binary and Configuration ID.

To counter this, BAB was introduced to prevent unilateral access: no single person could make an unapproved change to code in production. Over time, BAB's guarantee has expanded. It now ensures that all released software is free from tampering and its contents are well understood.

BAB policies ensure that:

- Code changes have been reviewed and approved by multiple people.
- The artifact was built on an approved build system hardened against tampering.
- The build was verifiable, meaning that the binary or package produced by the build can be verifiably audited by examining metadata which identify all source code used for the build.
- Metadata (provenance) was itself captured in a tamper evident way.[11]
- The configurations for jobs that host software services are also protected against unilateral modification.

Similar to a very detailed SBOM, this provides deep visibility into the source dependencies used to create an artifact in production. Additionally, the isolation and cryptographic signing requirements imposed by BAB upon Google's build systems enable us to trust that the artifact was not tampered with either during or after creation.

BAB is now used across Google to enforce technical policies around production integrity. By using technical means to enforce supply chain integrity policies at scale, Google reduces insider risk, promotes reliability and uniformity, and enables thousands of teams to comply with SDLC-related standards with minimal overhead.

[11] There's a difference between simply capturing provenance and capturing secure, tamper-evident provenance. We consider only the latter to be useful for supply chain protections. Secure provenance is generated at the time of an artifact's build, in an isolated environment on a trusted builder that is protected from interference. Cryptographic signatures ensure that the provenance is not tampered with after generation.

To enact technical change at large scale, we leaned on two key principles:

- We defined supply chain integrity as a ladder to climb, rather than an all-or-nothing "moonshot" effort. We gradually defined, implemented, and rolled out a series of smaller improvements that built on one another to create large reductions in attack surface over time.
- We worked with infrastructure owners across Google to implement "secure by default" systems. For example, Google's hosted build service now signs its builds with extensive provenance metadata by default, requiring no behavior change from users. We've also automated the policies which govern provenance verification and admission control on Borg, reducing the security burden for teams that deploy services at Google.

Supply-chain Levels for Software Artifacts

In 2021, we externalized BAB as SLSA, or Supply-chain Levels for Software Artifacts, so that other organizations could benefit from our decade of progress. Soon after, we donated SLSA to the Open Source Security Foundation (OpenSSF) for cross-industry collaboration. SLSA has been expanded from its Google-specific roots to now be useful to organizations and software producers of all sizes, including open source software producers and project maintainers.

Like BAB, SLSA is organized into well-defined levels, each laying out increasingly rigorous security expectations. This is intentional to enable systematic organizational change: a given team, department, or operational domain can begin scaling the ladder at any point, with each rung reachable from the previous.

When using BAB to secure Google's production systems, this progression of levels provided an essential structure for transforming the maturity of security controls.

Similarly, SLSA provides a clear roadmap with benchmarks, assessments, and goals. The leveled structure makes complex changes more manageable by defining clear standards and achievable steps to reach them. The framework also creates a common vocabulary and context for change initiatives. This fosters clear communication and alignment across teams and departments about expectations and progress.

This latter point—shared language—speaks in turn to a major benefit brought by the SLSA framework in the industry at large. As well as outlining supply chain best practices, and enabling automated policy to manage and mitigate risk, SLSA has provided a shared vocabulary and conceptual model for the problem space as a whole, which is applicable in turn to the AI supply chain space.

Model provenance

Across Google's first-party and open source AI development ecosystems, we're in the process of adopting the SLSA framework and format to sign **model provenance**.

This metadata document cryptographically binds a model to the **service account**—an identifying account that represents an application rather than a human user—that was used to train it. It will also enable Google to verify all of its models against the expected signing keys, such that an insider cannot overwrite or change the model (including the weights that determine its behavior) without detection.

Achieving higher SLSA build levels is particularly challenging for models compared to other software types, because training is long, resource-intensive, and difficult to sandbox. We're currently exploring approaches to harvest provenance from Google's existing comprehensive data-access logging infrastructure, instead of relying solely on AI platforms to provide the entire bill of materials.

This will enable us to extend the signed SLSA provenance certificate to include additional metadata about the data used to train the model: **data provenance.**

Data provenance

For complete understanding of what is in a model, we must also understand the data that was used to train it. Data provenance entails recording the source of all data examples used during training and evaluation of models. If data is transformed prior to training (e.g., for data cleaning purposes), we recommend building a separate dataset with its own associated provenance document, linking to the original data.

Currently, data provenance does not convey the supply chain *integrity* of data. As mentioned earlier, we are investigating ways to annotate datasets with their own signatures, which would provide the integrity to know whether they had been tampered with after signing. We currently have open questions about the granularity of record keeping: we could just record the datasets by a URI, or we could cryptographically encode every record in the dataset into a structure that would allow detecting if a specific example has been used during training or not. There are also possibilities between the two extremes, and we are analyzing trade-offs along the spectrum.

Within Google, we provide tooling for data and model cataloging to improve both the distribution and discovery of assets developed and used by teams for AI model development and deployment. The data and model catalog is focused on the snapshot and version of metadata, and the data or model's properties and attributes. Entries in the catalog are closely associated with the underlying asset that is stored in one or many locations, which in turn helps users short-list the available assets that meet their criteria for the AI project being worked on, request access and compliance approvals, and start using the asset immediately thereafter.

Every dataset version and model version is also issued a globally unique entry ID that can be used as part of the many data and model loading libraries we support within Google. Using the IDs, we are able to associate the lineage of downstream jobs and tasks back to the exact version or snapshot of a dataset or model ingested.

Within Google, we record the following asset metadata for models and datasets:

Model Attributes	Data Attributes
Name	Name
Unique Global Version Identifier	Unique Global Version Identifier
Date Created	Date Created
Date Updated	Date Updated
Format	Format
Location	Location
Size	Classification results (what the data is comprised of)
Owner	Owner
Schema	Schema
Documentation	Documentation
SHA-256 Hash	SHA-256 Hash

Provenance collection is often an afterthought for an organization but we have learned that it is difficult to retrofit this into model training workflows. As a first step, it is important to have rigorous bookkeeping for the artifacts themselves. Every model and dataset needs to have globally unique identifiers and need to be immutably versioned as a foundation for provenance collection. Approaches that can reliably record provenance include:

- **Explicit Provenance Logging:** Recording lineage relationships in I/O libraries such as data ingestion or model checkpointing libraries. The challenge is achieving comprehensive coverage over all the ways in which datasets and models can be read or written.
- **Infrastructure Log Harvesting:** Low-level infrastructure logs that universally record all file accesses can be mined for AI artifact provenance, but it is often hard to accurately relate low-level logs back to models and datasets.
- **Execution Sandbox:** Ideally, AI workflows like training or data enrichment jobs provide a manifest of inputs and outputs and a sandbox restricts any access outside of the manifest while recording every input and output. This requires organizational mandates for all workflows to be run within such an execution sandbox.

Once an organization has provenance for all steps in the AI supply chain, they can construct policies that can be evaluated automatically to detect if a model is acceptable to be used in production or for training a new model. This results in a shift-left approach to model development, as usage of bad models or datasets can be detected earlier in the development lifecycle instead of at launch approval stage, after costly training and testing.

Solutions for third-party AI model development

Google invests significantly in securing the open source software world, including support for SLSA and the Sigstore project. In 2023, we open sourced our work to apply existing solutions such as SLSA and Sigstore to AI.

In this section we are focusing on open source solutions such as SLSA and Sigstore for models that are developed by authors that are in a different organization than the developers of the software system that uses the models. These can be open models (models whose weights are available to end users), or fully open source models (both the weights and the training source code are available). In either case, when working with these models, we should adopt a more stringent security posture in which we consider the model publisher as untrusted. We are exploring the different types of claims that can be made about the models, and how these types of claims can be publicly verified.

Signing outside Google

For signing open models, we recommend Sigstore, a free, public-good service that simplifies code signing and verification by abstracting away most key-management complexity. Sigstore's public signing ledgers form the foundation of trust for SLSA provenance (introduced earlier in this paper). Sigstore can be used to sign artifacts after building them. Users can then verify the signature, proving that the artifact has not been tampered with since it was built. Since Sigstore's signing process supports using temporary or "ephemeral" keys that expire shortly after the signing event, there is no ongoing burden for developers to maintain or rotate keys long term. This addresses the risk of a later key compromise that would negate the signature. Like SLSA, Sigstore's benefits will also be applicable to the AI supply chain space.

Most immediately, we are considering how models can be signed. This can be seen as a claim made by the model publisher that the user is seeing the authoritative version of that model—the one intended to be released by the publisher. Note that we can make a distinction between the publisher of a model (the identity uploading the model to the external model hub) and the trainer of the model (the identity that triggers the training job that produces the model). The threat model we are adopting here may assume that the trainer might not trust the publisher or the internal storage system and might want to protect the integrity of the model during these steps.

For these purposes, we have a library for signing models with Sigstore. Depending on the threat model, the library supports signing just the final model or checkpoints during training. For long training jobs running on shared environments, it is preferable to sign all checkpoints so that attackers cannot force loading from a corrupted checkpoint after a training job is interrupted. However, we are aware that checkpoints can be huge and the signing process can add delays, so checkpoint signing is not active by default. We are exploring parallelization approaches for the signing process, so that integrity protections can be deployed without large latency/efficiency costs.

We are planning to integrate the signing library with model training frameworks to enable transparent signing of models without asking model trainers to update their code. We are also integrating the library with model hubs to allow signing the model when it is uploaded. The difference between these two modes is that, while signing as early as possible ensures integrity for a larger part of the supply chain, not all model training frameworks might support signing. Thus, rather than refusing to upload unsigned models, model hubs have the opportunity to offer signing just before upload.

Once a model is uploaded, the model hub can verify its signature and display a specific label on a **model card**—a public source of information for models that's similar to a `README.md` file in code repositories. These short documents are attached to models and used to describe what the model is, how it was trained, its use cases, and other useful information. A label indicating a verified signature in a model card signals to users that the model has integrity protections, and security conscious users can also validate the signature themselves before loading the model in production. We recommend that model verification is performed, to cover the risk of the model hub itself being compromised.

Furthermore, we are planning to also offer signing for datasets to reduce the opportunity window for data poisoning. This would strengthen other mitigations already implemented by data storage systems.

Integrity outside Google

While artifact integrity helps in detecting artifact tampering after the artifact been built, in the longer term, we are also considering verifiable claims that can be made about the model's training pipeline. We are exploring building control features into existing training environments that would allow the model publisher to output verifiable provenance about the model (e.g., its code dependencies, the training environment, and even the training data or pre-trained models that were used). This is where SLSA for models comes into play.

However, in order to achieve a high level of SLSA assurance, the provenance generator must run in an environment that is not controlled by the person triggering the build. This means that we need custom *reusable* trainers for AI models. These must support hardware accelerators, a requirement that is usually not present in the traditional software world.

We are currently working on extending the SLSA standards to include a separate dependency track (to enable reduction of risk arising from using external dependencies) and a separate source track (to provide protection against tampering prior to the build).

With the provenance in place, we are also exploring more advanced functionalities, such as verifiable claims around the inclusion or non-inclusion of a specific artifact in the training data.

To begin with, we are considering the use of the Croissant format for describing the datasets. While this was developed to streamline the discovery and understanding of external datasets, we can use it as the canonicalization layer needed to ensure that computing the digest of a dataset can be done in a deterministic manner.

As we mentioned earlier, a significant component of the lifecycle of LLMs is represented by post-training evaluations. Currently, these are performed without security guarantees, relying on implicit trust on the evaluator. We are thinking of performing these evaluations on trusted enclaves, reporting the benchmark scores in a tamper-proof way, using the in-toto attestation framework.

Finally, we need to consider the usage reviews when open models are involved. The most common scenario is when a new AI-powered application is awaiting launch approvals, but we are also thinking about how libraries that use AI models can be safely imported into a company's internal source control. In either of these cases, we need a way to look at all the associated supply chain metadata (generated based on solutions presented before in this section), and combine them into information that can be both human and machine readable. This would enable faster reviews, as well as automating and shifting left of approvals for using open models.

For this part of the problem space, we recommend using Graph for Understanding Artifact Composition (GUAC), as it is the system of choice to understand large supply chains both in open source and at the boundary between a company's internal systems and open source software. With minimal changes, GUAC can be used as a window into the AI model development lifecycle, as a tool for aggregating and querying metadata across the AI supply chain.

Open questions for industry consensus

Even given all the solutions already at hand for AI supply chain security, open questions remain and problems still need to be solved. While we've presented some of our opinions above, we believe it would be fruitful to come together as an industry to find common consensus on the following open questions in particular:

- What information should model and data provenance contain?
- What format should be standardized for model and data provenance for interoperability?
- How and where should model and data provenance be stored, shared, and verified?
- What level of detail should model and data provenance capture? For example, when recording dataset information in the provenance, do we commit only to the name of the dataset or to each individual example in the data? The latter would allow answering questions regarding whether a specific piece of information was used during the training process, but it would be larger and require more effort to generate.
- Should datasets be signed for integrity? If so, are there additional data-specific fields to add to the SLSA standard?
- How can we protect model training against consuming untrusted sources or tampering with provenance before it's been signed?[12] Full sandboxing or resource isolation seems difficult and costly, so are there alternative mechanisms to explore?
- Should model hubs support provenance verification as a built-in feature?
- Should the industry embrace ML-BOM[13] or provenance?

[12] Full sandboxing or resource isolation is important at higher SLSA levels to ensure that provenance is protected from tampering during generation. To apply the same sandboxing to the training process seems difficult and costly. We are considering whether there are alternative methods to explore, such as reducing the burden of sandboxing during the training process by leveraging observability into dataset storage system logs as a supplementary mechanism for fleshing out a provenance document ("log harvesting").

[13] There is an industry discussion about extending the SBOM concept to a new ML-BOM. However, we feel that the SLSA provenance can capture the same information about the supply chain, while the remaining information in an ML-BOM is typically present in a model card. This view might change in the future, however, and we are looking at relevant industry developments and will adapt as needed.

04

Guidance for practitioners

Each organization's approach to supply chain security will look different, based on internal processes and platforms.

But even if the approach to capturing and handling the metadata needed for supply chain insights will vary, the ultimate goal will likely look similar: to build your supply chain with insight and transparency. We suggest reviewing the Guiding Principles section earlier in this paper about the technical considerations to achieve this goal.

An organization will be in a good position if it's able to do the following for all software and AI artifacts:

Capture metadata

Capture enough metadata to understand the lineage of each artifact (models or otherwise!). You want to be able to answer basic questions: where an artifact came from; who authored it, changed, or trained it; what datasets were used in training it; and what source code was used to generate the artifact.

Organize metadata

Organize the information to support queries and controls. In the event of an incident, you'll be able to know the blast radius of affected components. At launch, you can enact appropriate governance. During development, controls will determine whether artifacts meet guidelines for use.

Increase integrity

With time, work toward increasing integrity of both artifacts and associated metadata. Ultimately, the metadata should be captured during the artifact's creation in a non-modifiable, tamper-evident way. The artifact itself should be cryptographically signed using next-generation signing techniques that reduce the burden of key management, to show whether an artifact has been tampered with after the fact.

Share with others

Finally, as a best practice, share the metadata you capture in an SBOM, provenance document, model card, or some other vehicle that will assist other developers. This type of transparency into a model's creation increases trust and assists in tracing unexpected behavior from a model back through a complex supply chain to discover the source of the problem.

Conclusions

Our approaches outlined in this white paper are intended to guide industries and organizations seeking to secure their AI software supply chains. We believe that extending existing software supply chain solutions is an effective way to counter many of the risks associated with AI software supply chains. Rather than creating new solutions, we can approach AI models like traditional software. By diligently applying established software supply chain security practices and carefully tracking datasets, organizations can bolster their defenses against malicious attacks and recover more quickly from unintended vulnerabilities.

The stakes are high. AI systems are increasingly involved in sectors ranging from healthcare to finance and infrastructure. Over the last decade we've seen in traditional software domains that you are only as secure as your weakest link, and the weakest link is often an overlooked piece of a supply chain. By applying the solutions laid out in this paper, we believe that we can collectively strengthen the links that tie the AI software supply chain together.

Collective action is key. As with traditional software, no AI model is an island. No matter how self-reliant an organization is, there will always be dependencies, datasets, and other shared components involved. By increasing the information we capture and share about these components, we can secure the fundamental building blocks of the shared AI ecosystems and help secure the AI software supply chains for everyone.