CS 453/698: Software and Systems Security

Module: Usable Security

Lecture: Software Supply Chain Security

Adam Caulfield

University of Waterloo

Spring 2025

Reminders & Recap

Reminders:

- A3 is due on July 11
- Send your research project proposals to Meng and me!

Recap – last time we covered:

Authentication

- Adversary & system model
- Password protocols
- Alternative methods

Attestation

- Adversary & system model
- Example protocols

Today

Software Supply Chain Security

What is the software supply chain?

Attack vectors within the software supply chain

Example safeguards and their target attack vectors

Proof of concept safeguard: in-toto framework

Software Supply Chain Security

Supply Chain Security:

Definition: *the security of the ecosystem, processes, people, organizations, and distributors involved in the development, manufacturing, and delivery of finished solutions or products.*

Finished solution/product is software → **Software Supply Chain Security**

Software Supply Chain Security

Software Supply Chain:

Definition: *a system of its participants with an interconnected set of resources and processes involved in the life cycle of software movement from the developer to the end user, namely, the design, development, manufacturing, supply, implementation, and support of programs and associated services.*

Can include various components & services

- Libraries, binaries
- Operating systems, package managers
- Compilers
- Development tools (e.g., IDEs, build systems, GitHub/Gitlab, CI/CD)

All make up the **SBOM: Software Bill of Materials**

- A list of ingredients that make up software components

Software Supply Chain Security

Key Characteristics of Software Supply Chain:

The goal:

- Deliver a software produce or service to end users
  - Platform-as-a-Service, Software-as-a-Service

Entities

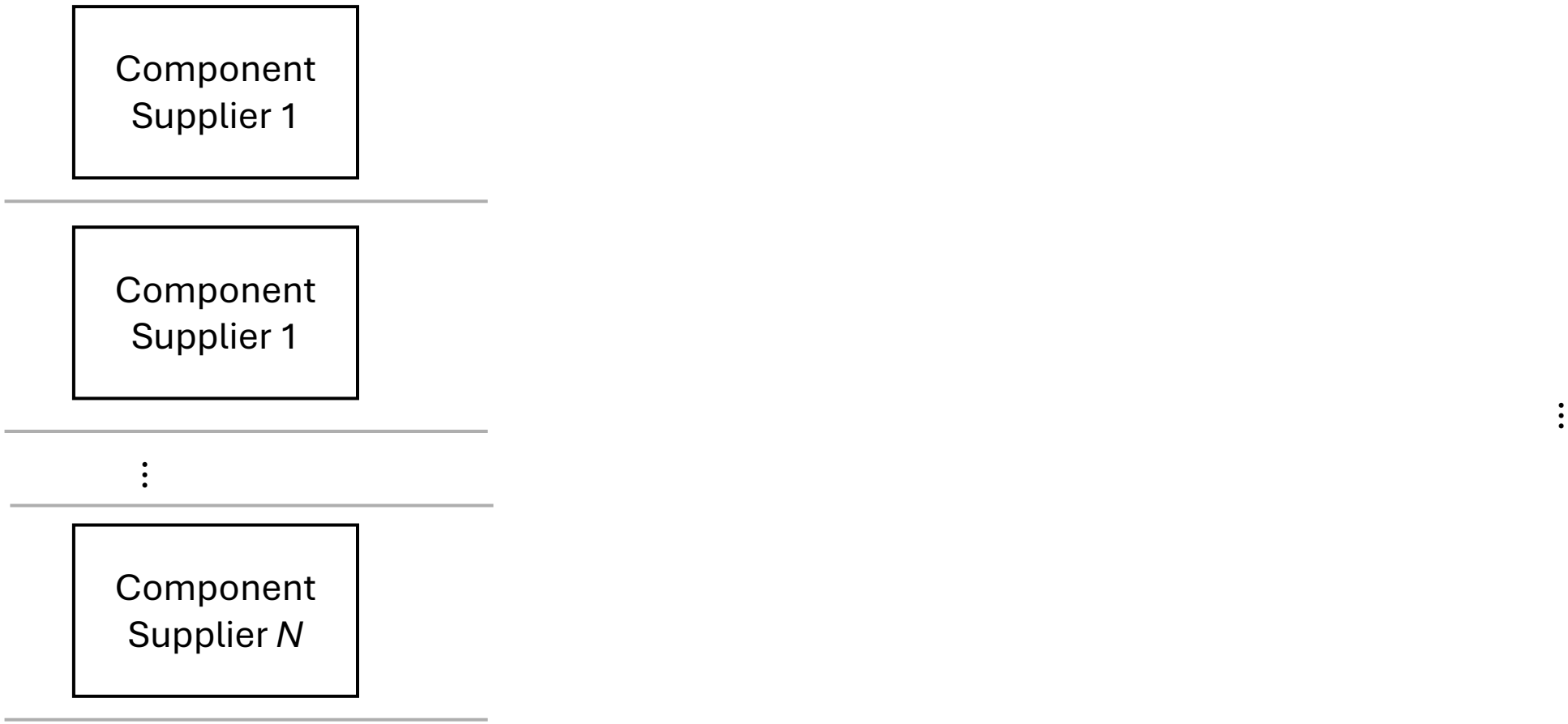
- Different organizations: developers, logistic, distribution, and assembly centers
- End users
- Might have an existing relationship (formal documented agreement)
- Can act as a supplier or customer

Two material/service streams:

- Upstream: streams related to product creation using third-party components
- Downstream: streams related to product delivery to end users

Software Supply Chain Models

(One) model of software supply chain structure

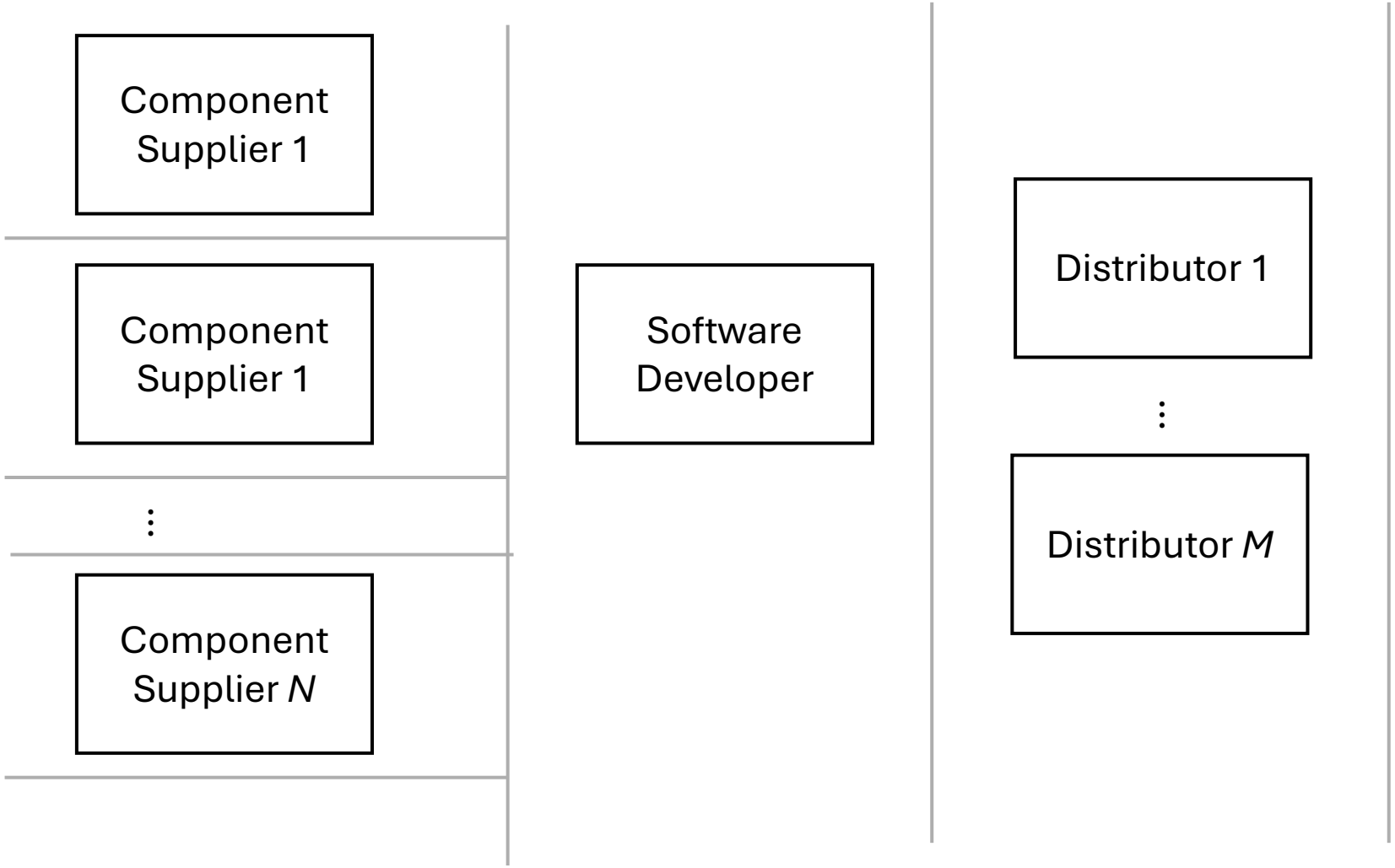


Software Supply Chain Models

(One) model of software supply chain structure

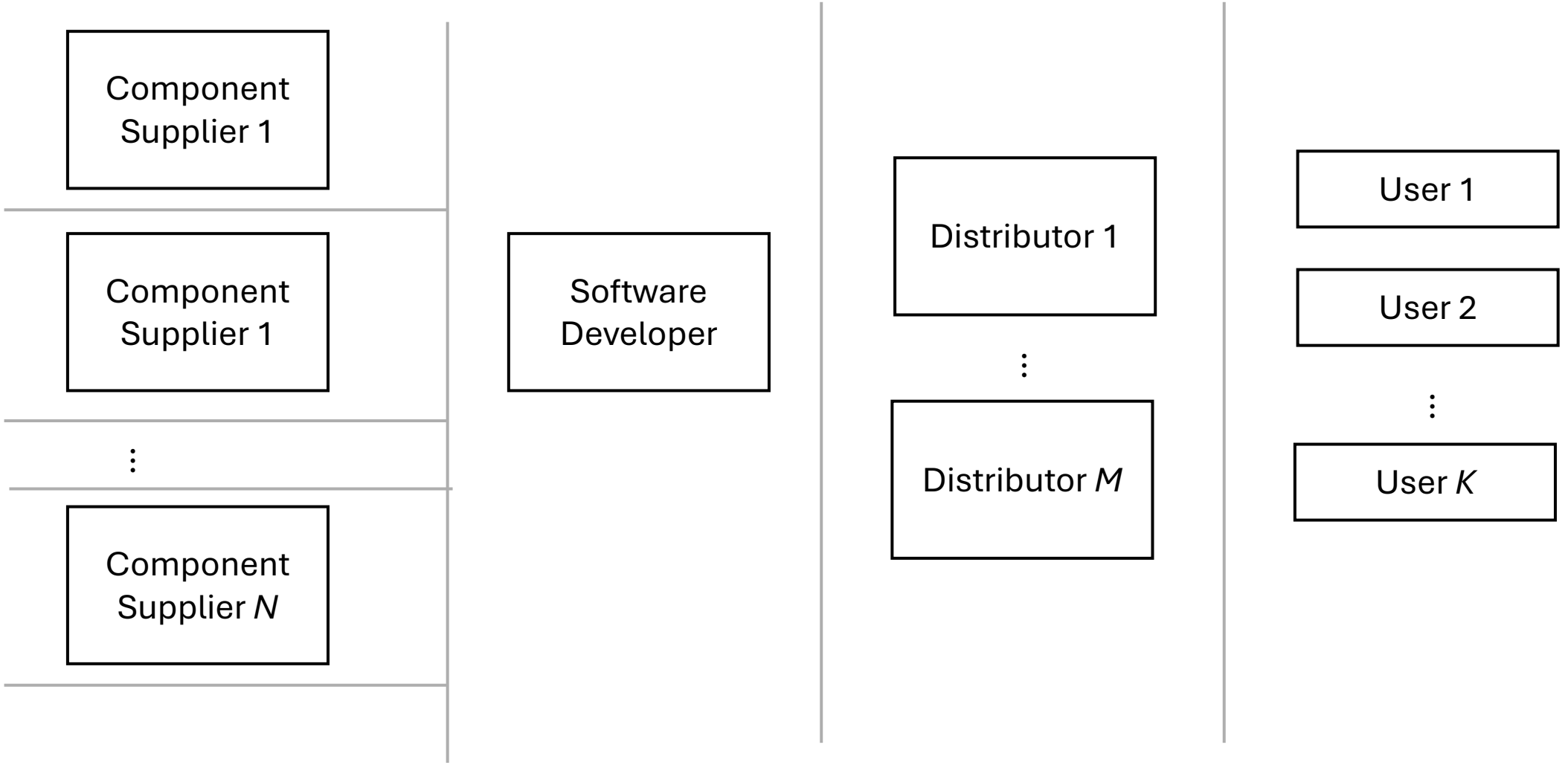
Software Supply Chain Models

(One) model of software supply chain structure



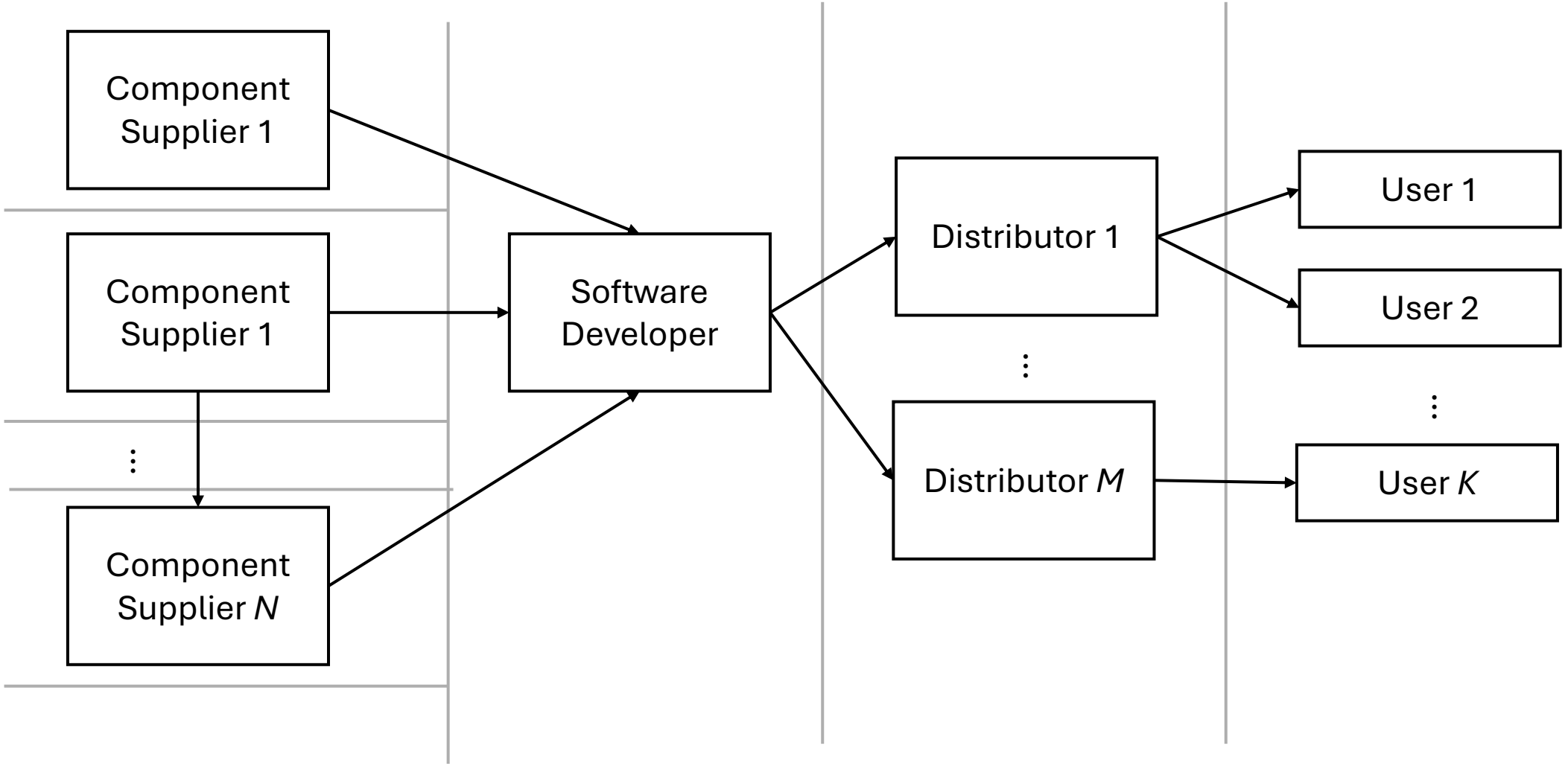
Software Supply Chain Models

(One) model of software supply chain structure



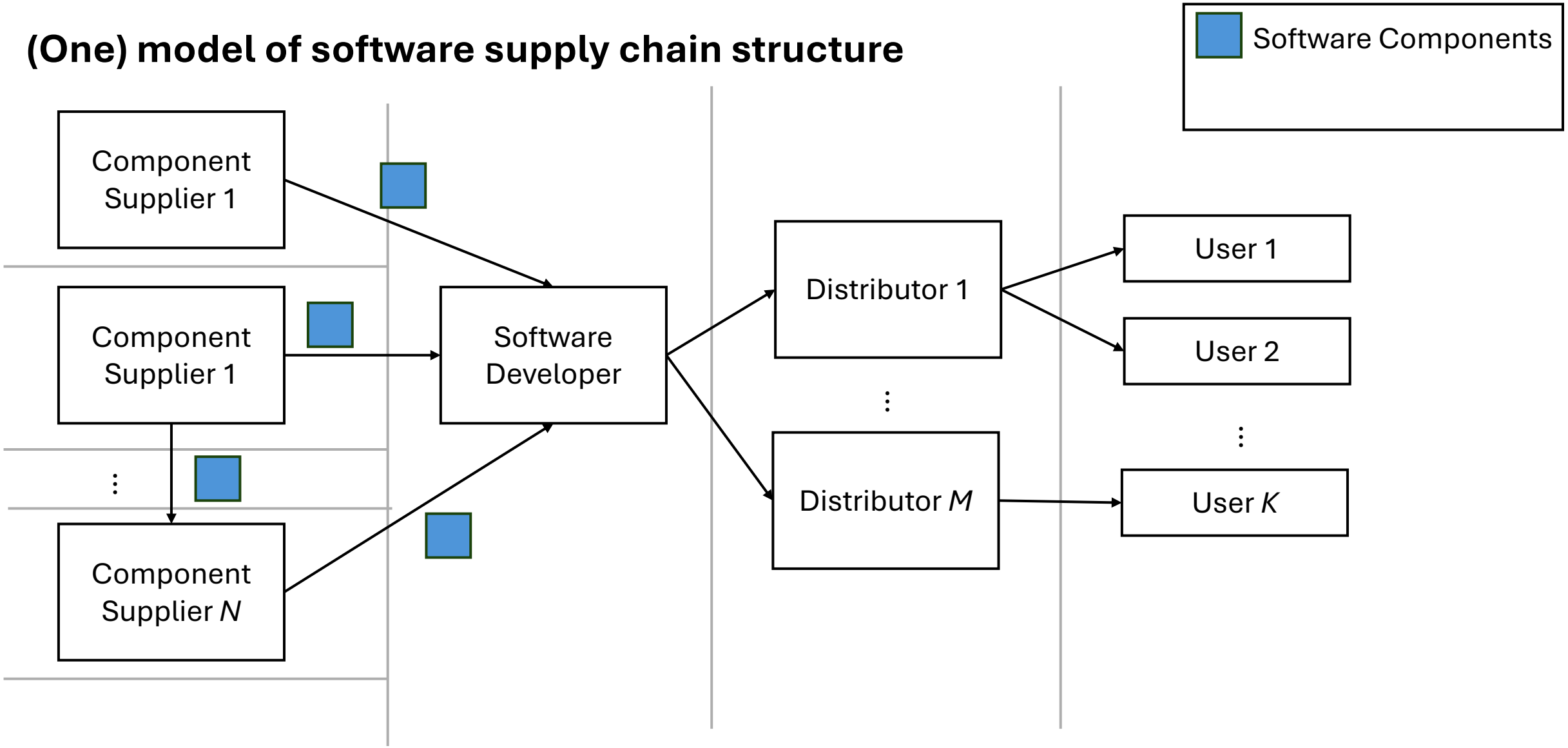
Software Supply Chain Models

(One) model of software supply chain structure



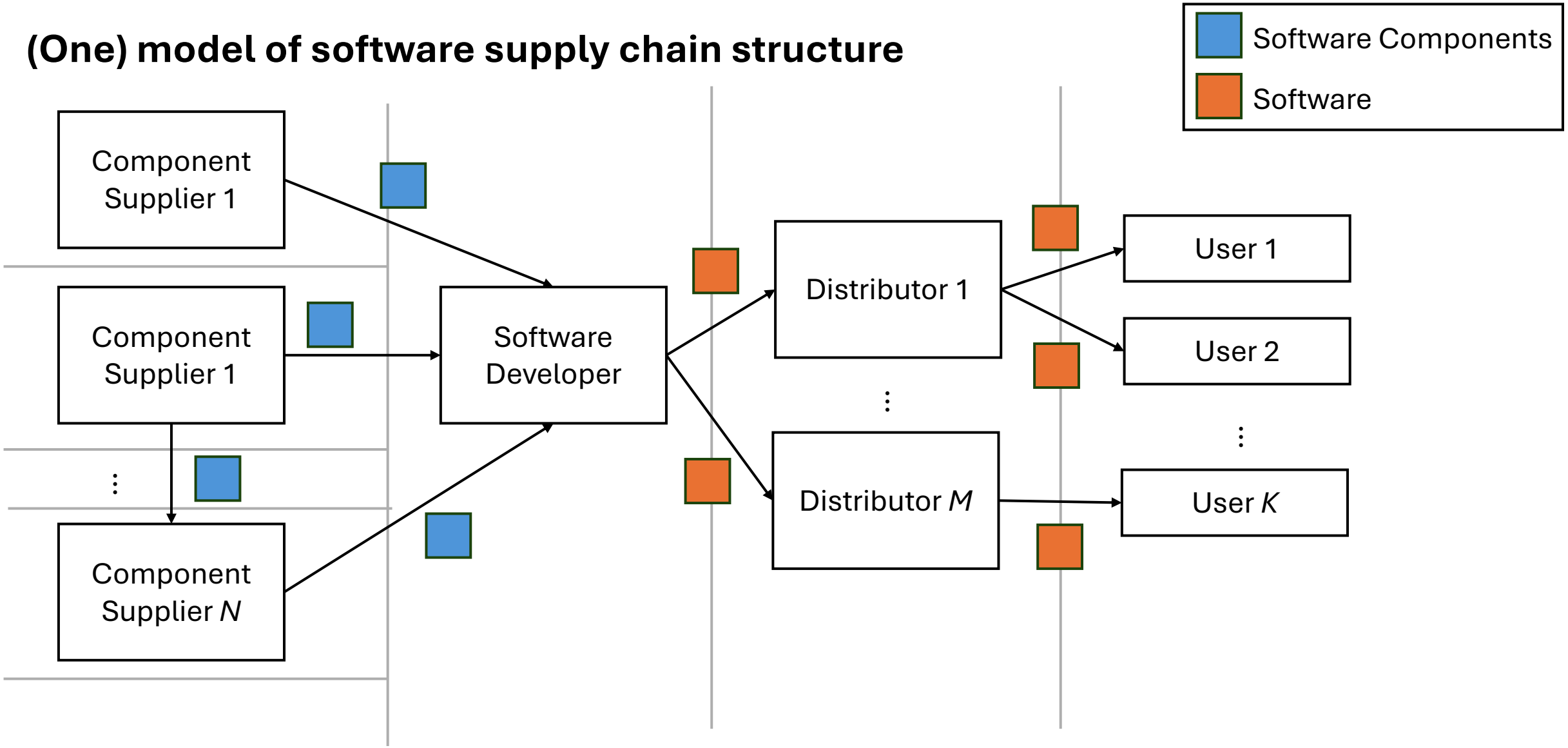
Software Supply Chain Models

(One) model of software supply chain structure



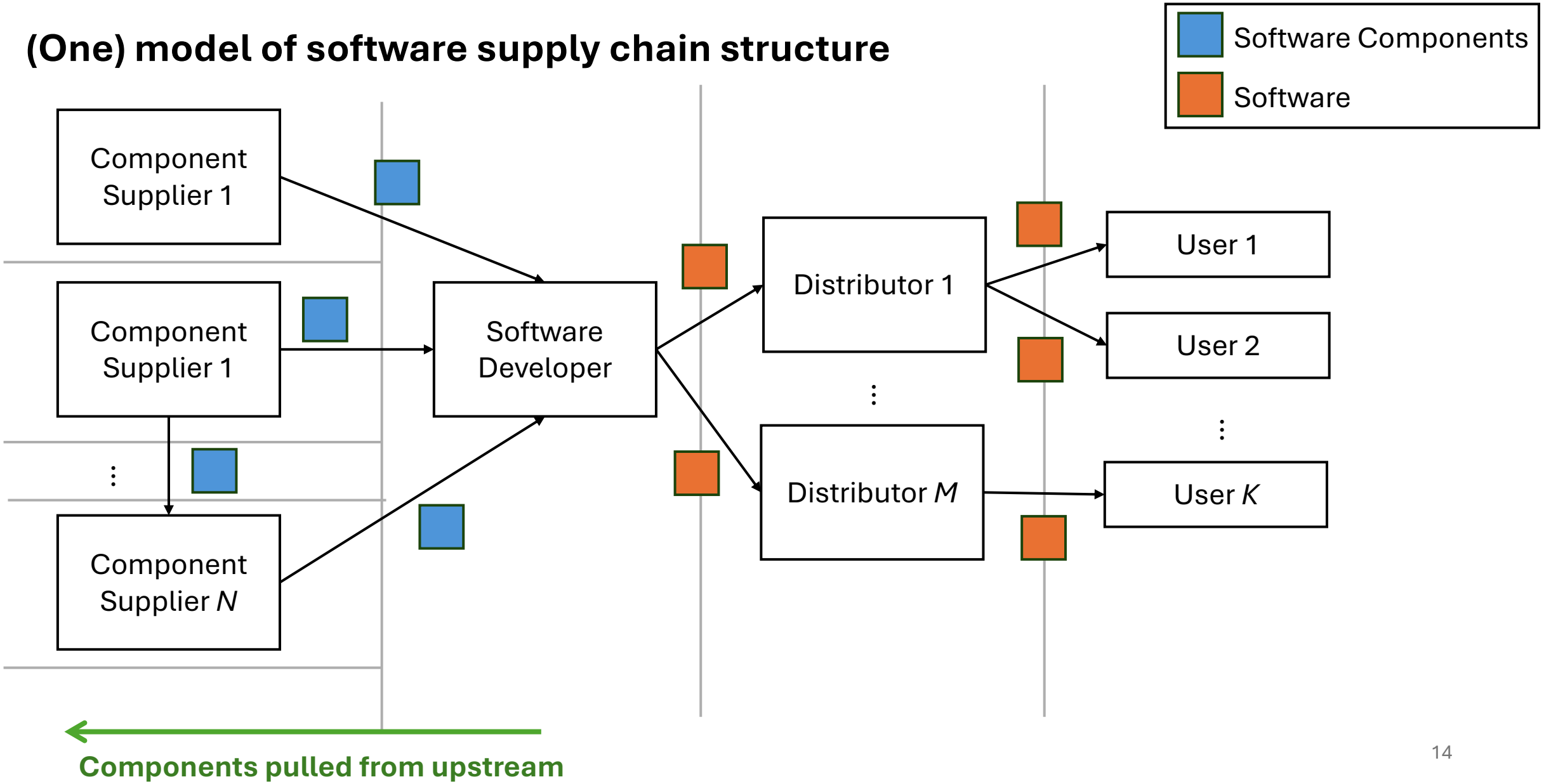
Software Supply Chain Models

(One) model of software supply chain structure



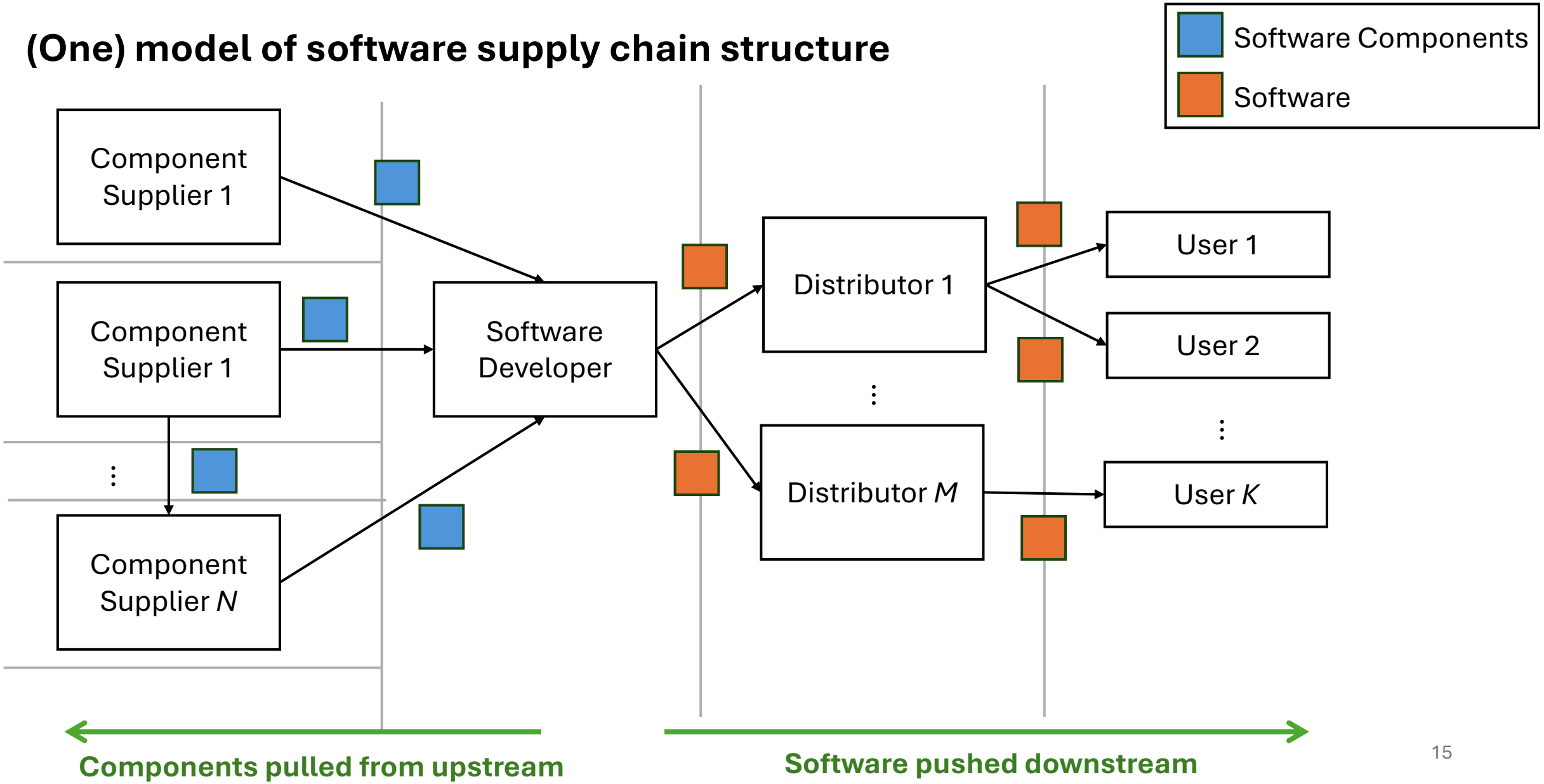
Software Supply Chain Models

(One) model of software supply chain structure



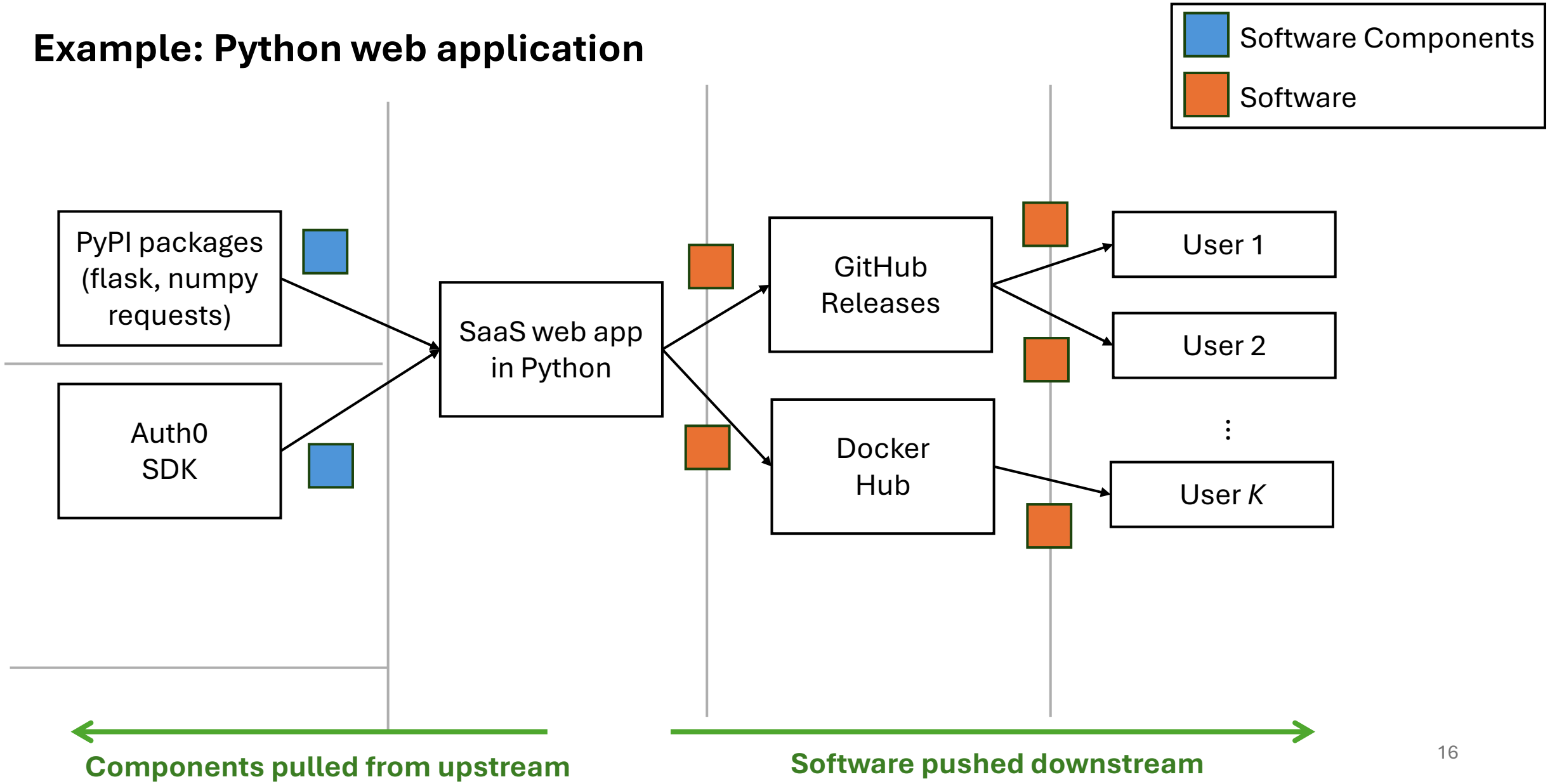
Software Supply Chain Models

(One) model of software supply chain structure



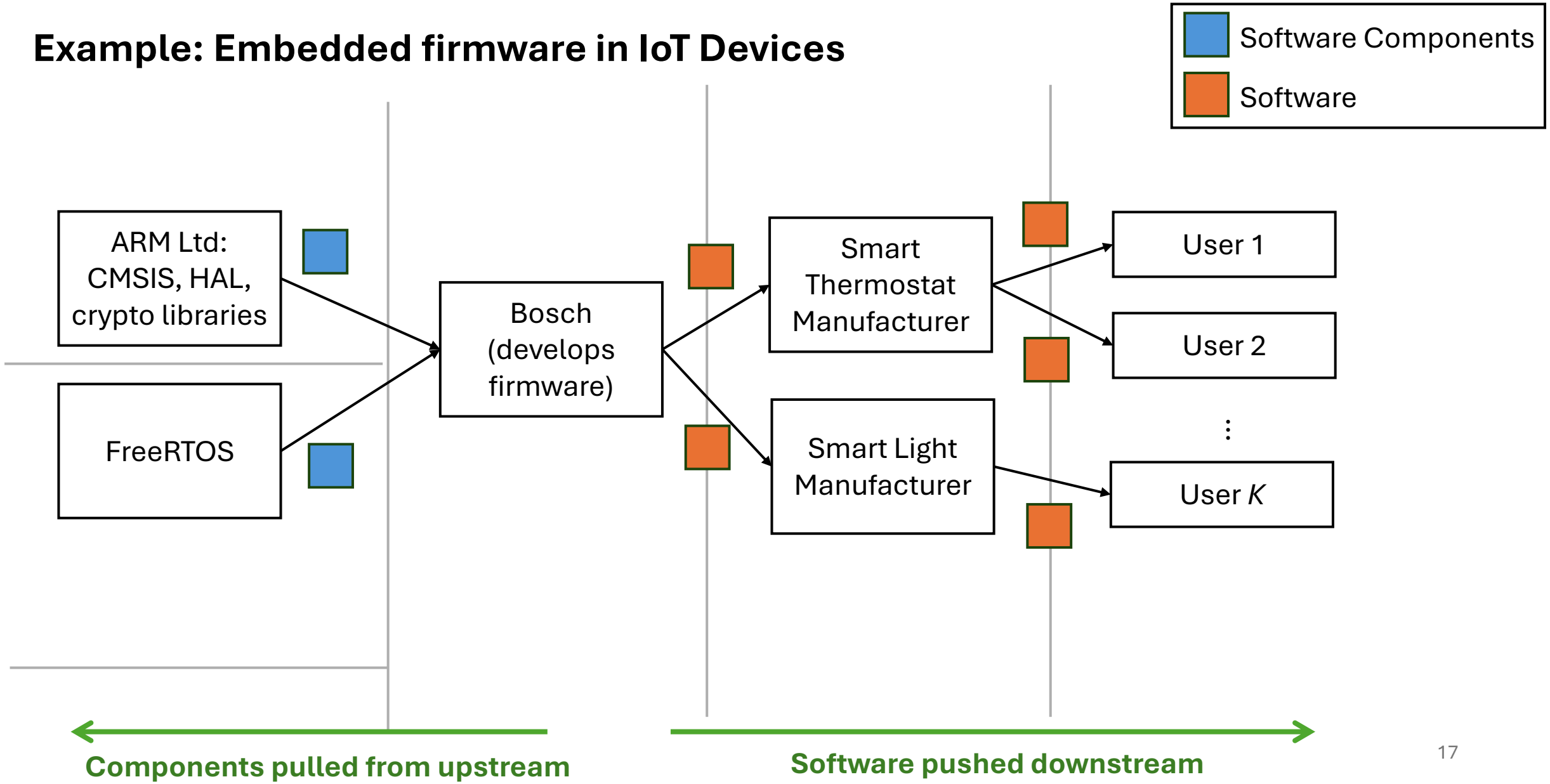
Software Supply Chain Models

Example: Python web application



Software Supply Chain Models

Example: Embedded firmware in IoT Devices



Software Supply Chain Models

Slowly becomes more and more complex…

Very hard to track

- Visualization tools: it-depends

Across organizations:

- Ecosystems Graphs of ML/AI software services

Software Supply Chain Models

What about with open-source software (OSS)?

Many modern OSS package ecosystems

- Enable deployment of OSS components
- Examples: NPM (JavaScript), PyPI (Python), Crates (Rust)
- Put heavy emphasis on re-use and sharing
- Developers quickly depend on hundreds of transitive dependencies
- Trust is sometimes implicitly placed in maintainers

Separate type of pipeline that feeds into creation software components…

Software Supply Chain Models

(One) high-level development and build model of OSS

Two classes of entities:

Maintainers 🗿

- Members of a development project who administer the depicted systems
- Provide review, approve contributions, define/trigger build processes

Contributors 🤠

- Submit code contributions (pull requests)
- Reviewed and merged into a project's code base by maintainers

Software Supply Chain Models

(One) high-level development and build model of OSS

Contributor

Maintainer

Software Supply Chain Models

(One) high-level development and build model of OSS

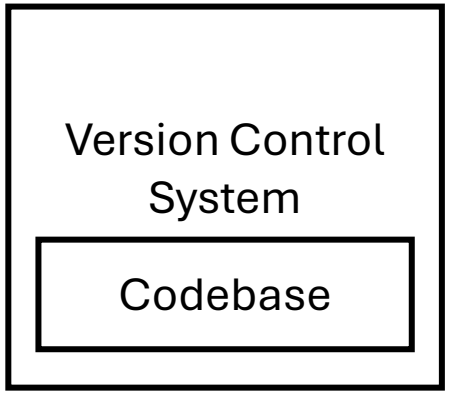
Core components:

Contributor

Maintainer

Version Control System:

- Holds a projects codebase & resources, tracks changes between versions



Software Supply Chain Models

(One) high-level development and build model of OSS

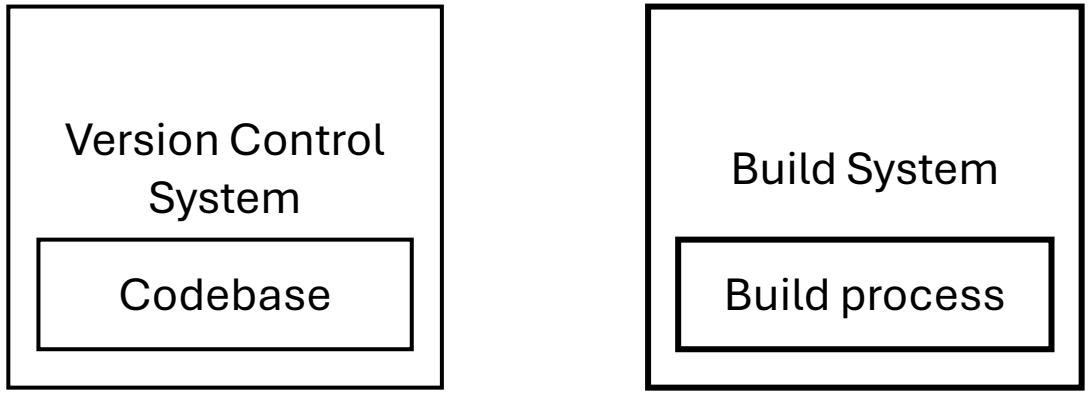
Core components:

Contributor

Maintainer

Build System:

- The system used for building the component / package
- Executes a *build process:* injects source code, produces dependency tree & artifacts



Software Supply Chain Models

(One) high-level development and build model of OSS

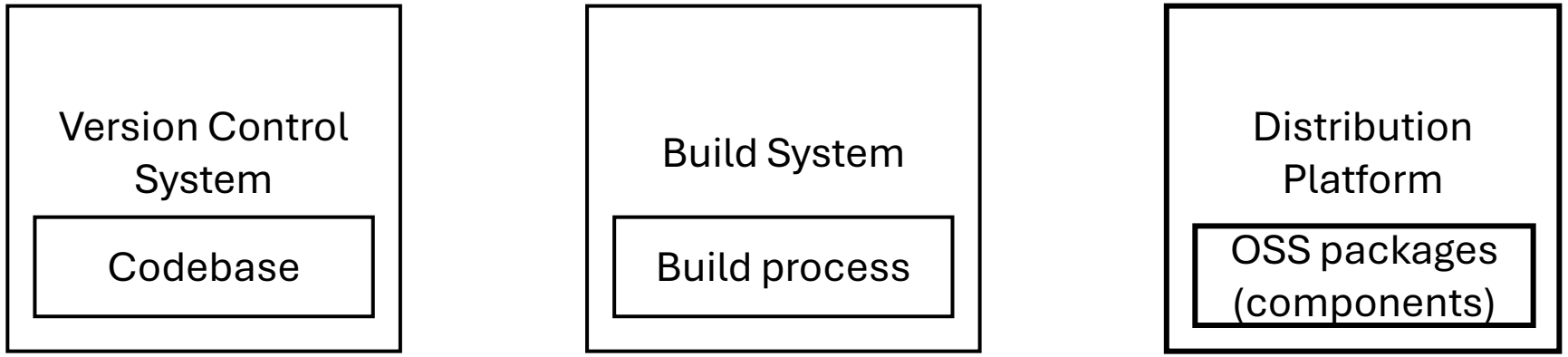
Core components:

Contributor

Maintainer

Distribution platform:

- Makes packages / components available (e.g., app stores, package repos)

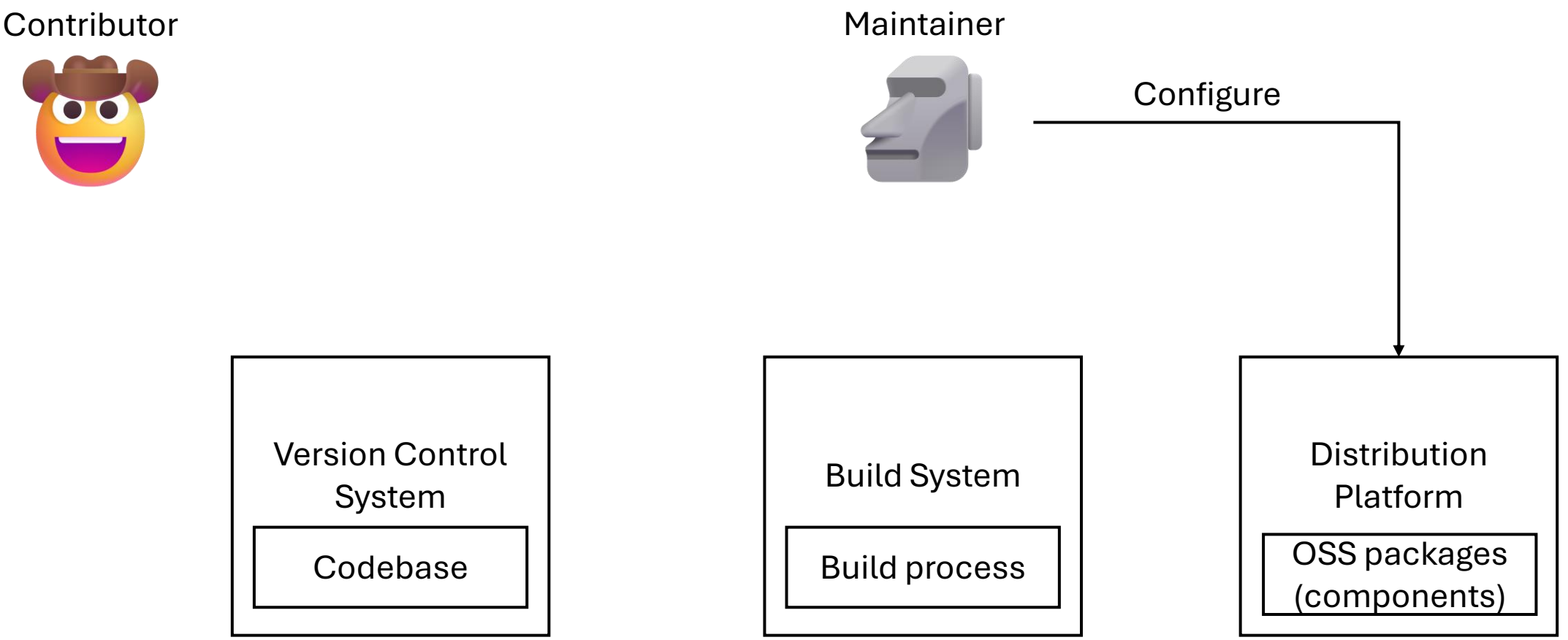


Software Supply Chain Models

(One) high-level development and build model of OSS

Maintainer configures the distribution platform

Setup Phase

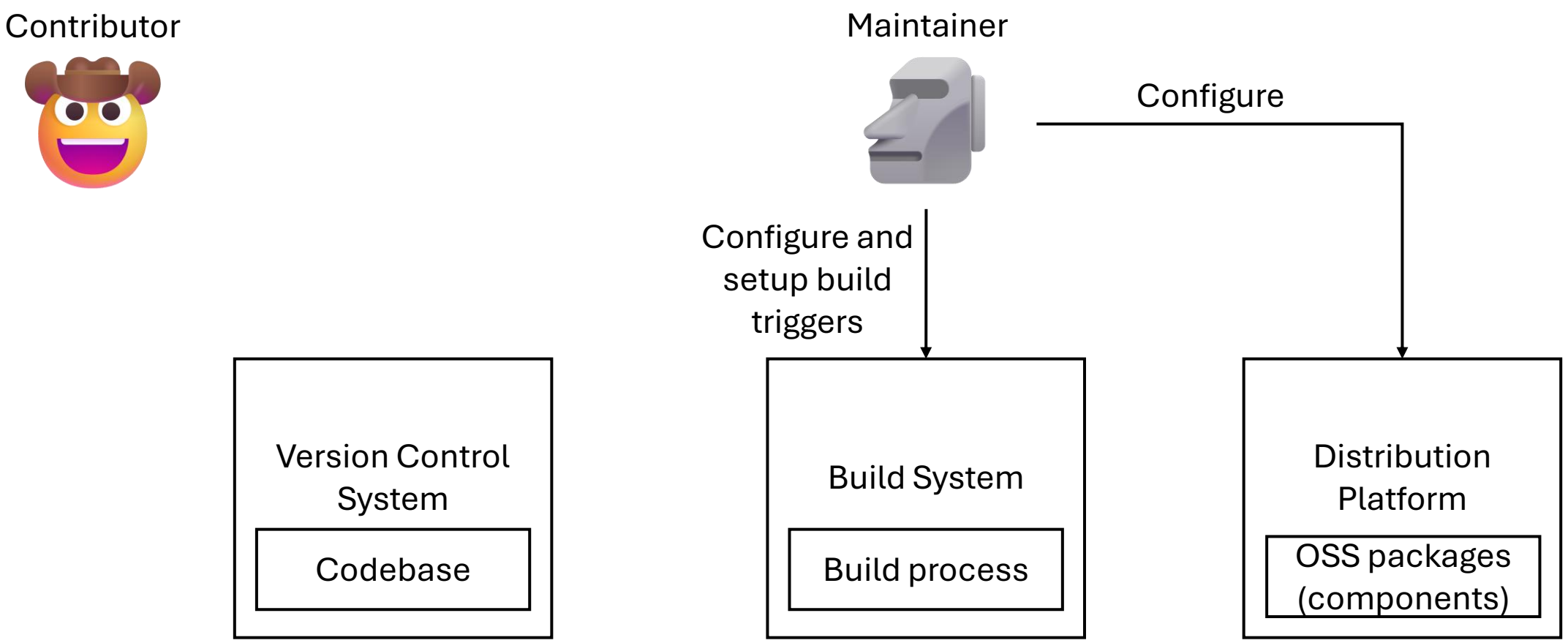


Software Supply Chain Models

(One) high-level development and build model of OSS

- Maintainer configures the distribution platform
- Maintainer configures the build system

Setup Phase

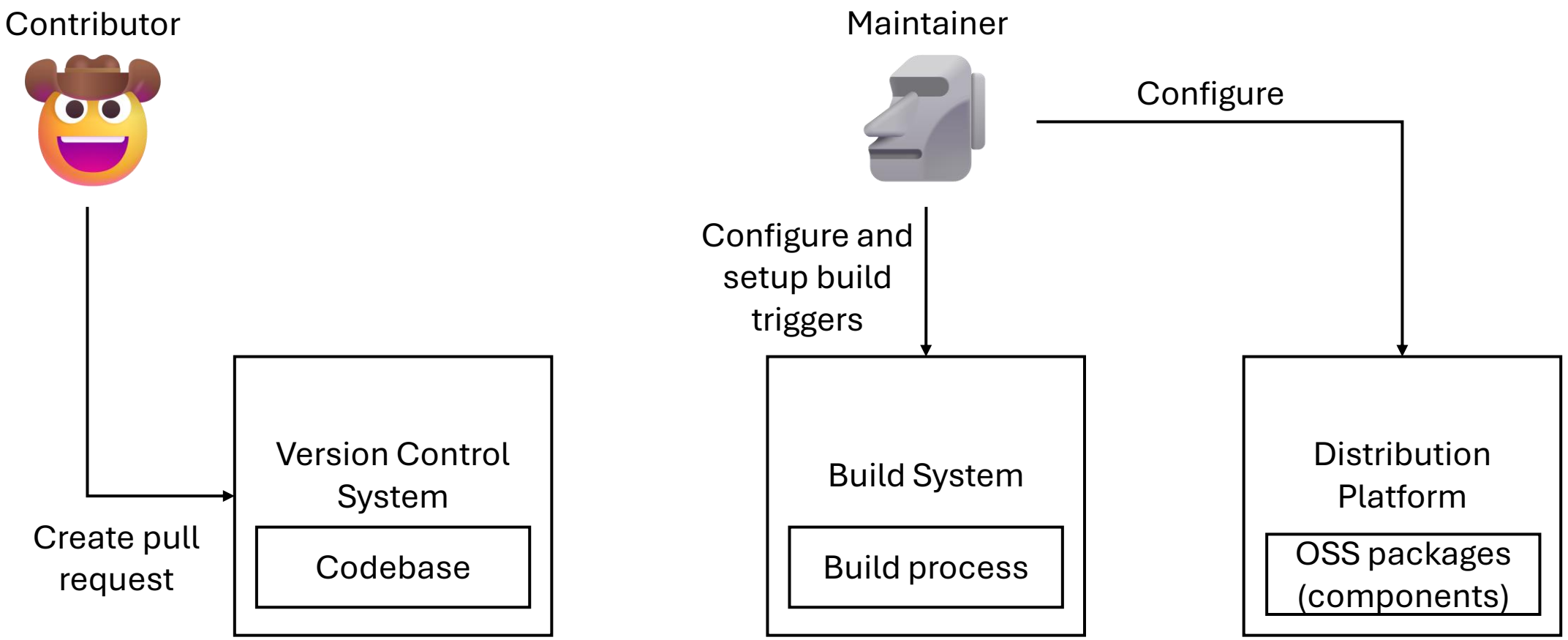


Software Supply Chain Models

(One) high-level development and build model of OSS

- Contributor submits code via a pull request to the VCS
- e.g., GitHub or other platform

Deployment Phase

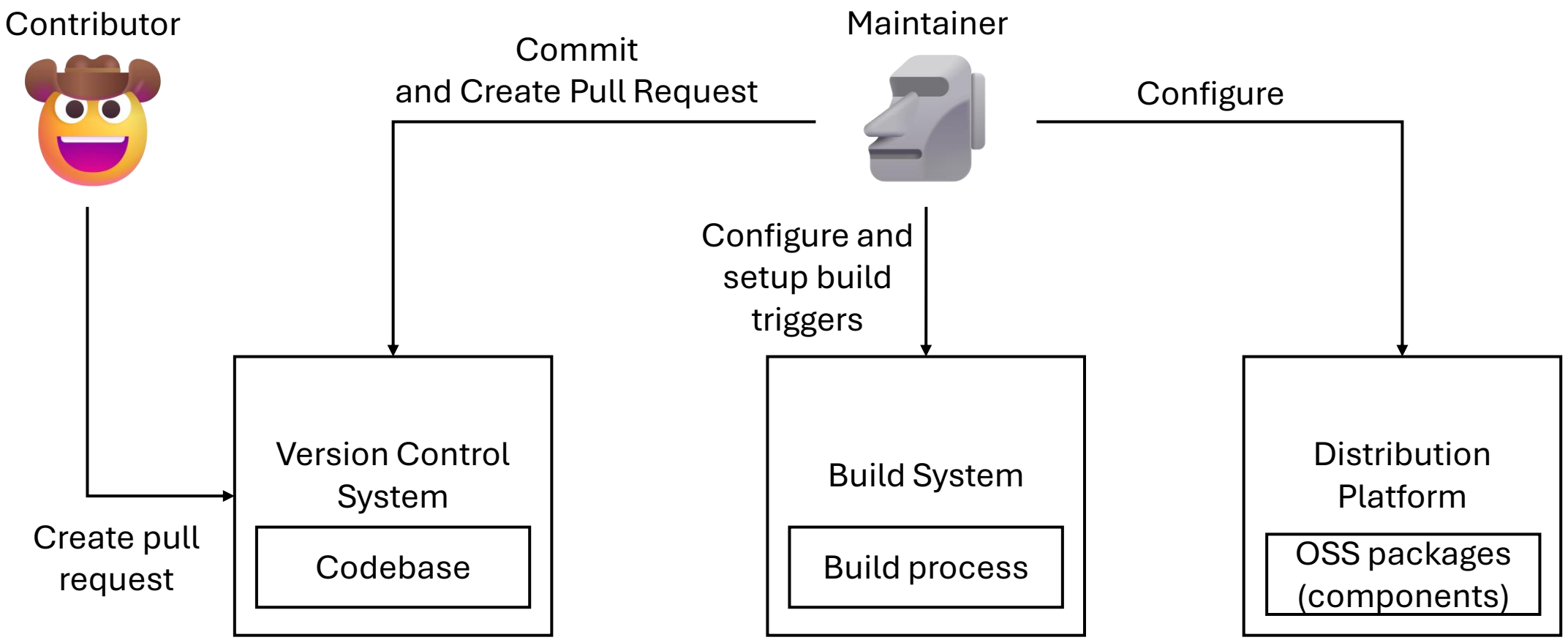


Software Supply Chain Models

(One) high-level development and build model of OSS

- Maintainer commits directly (small approved changes)
- Or creates their own PRs

Deployment Phase

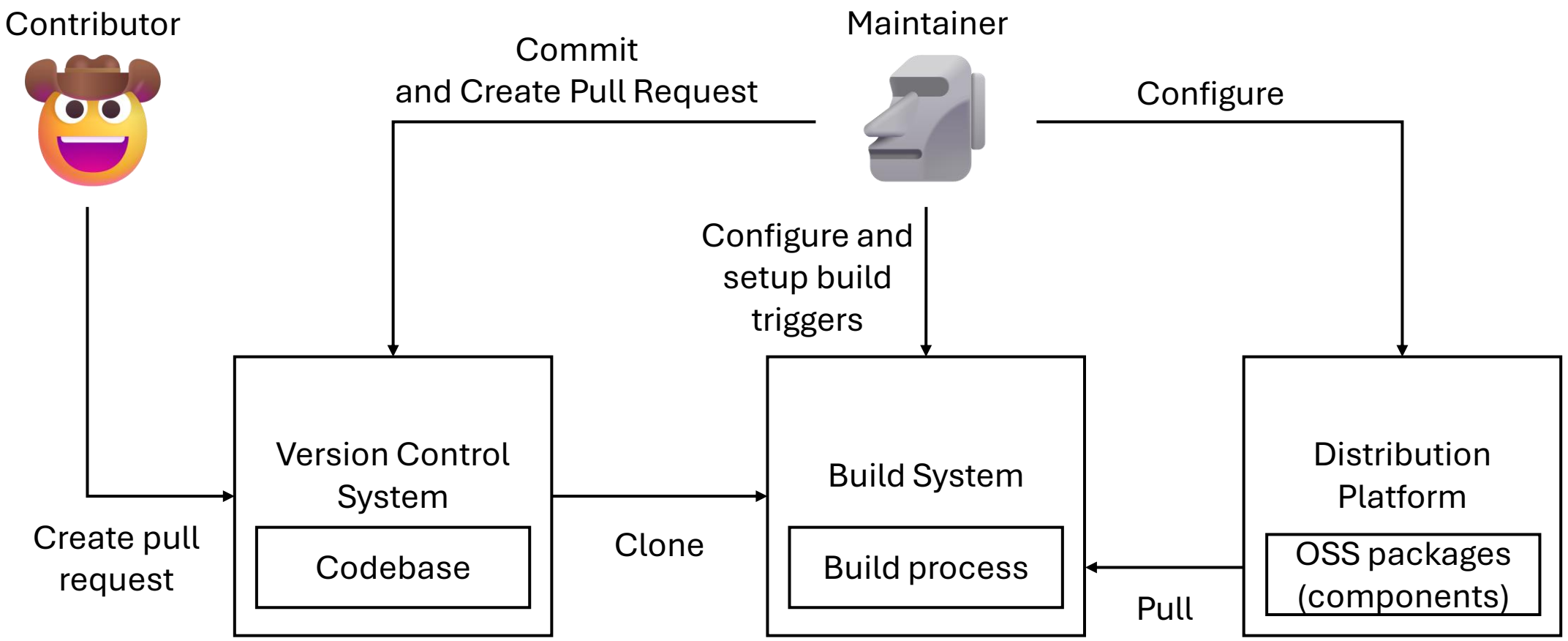


Software Supply Chain Models

(One) high-level development and build model of OSS

- Build system clones the current codebase
- Also, it pulls dependencies. Then, it executes its build process

Deployment Phase

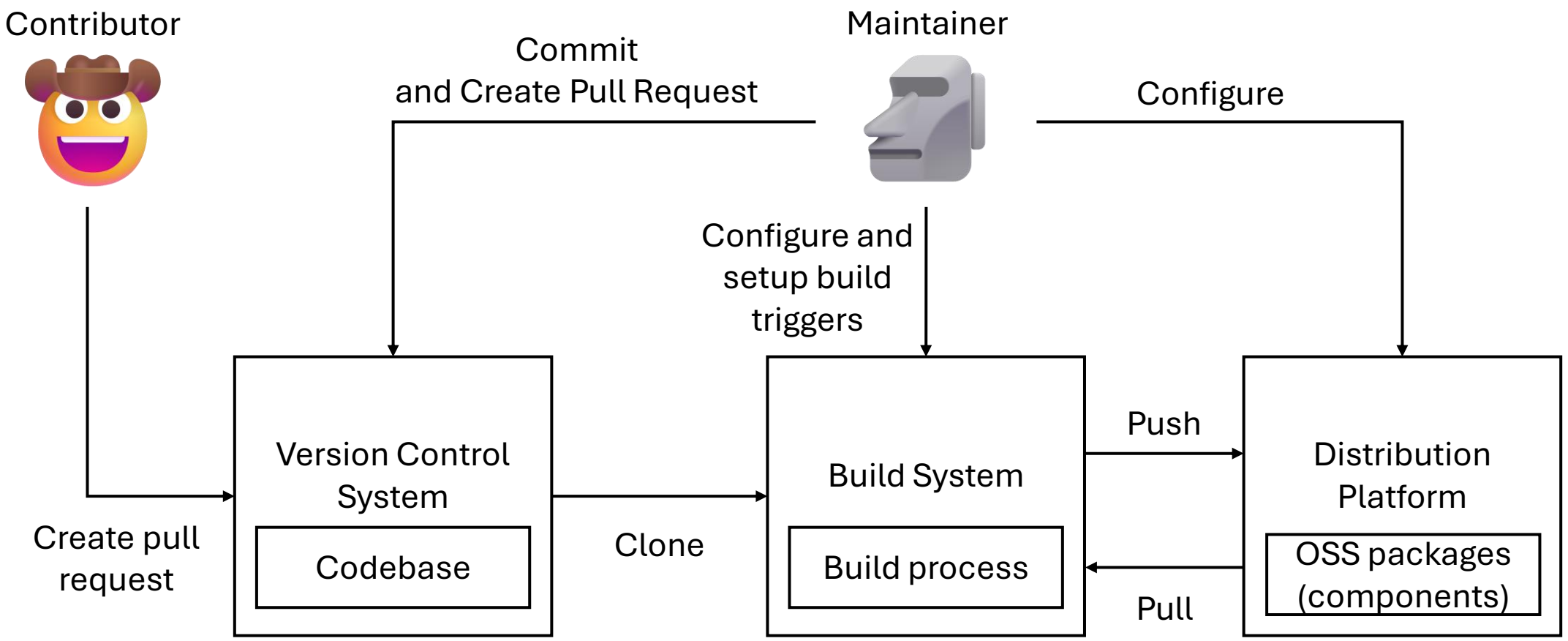


Software Supply Chain Models

(One) high-level development and build model of OSS

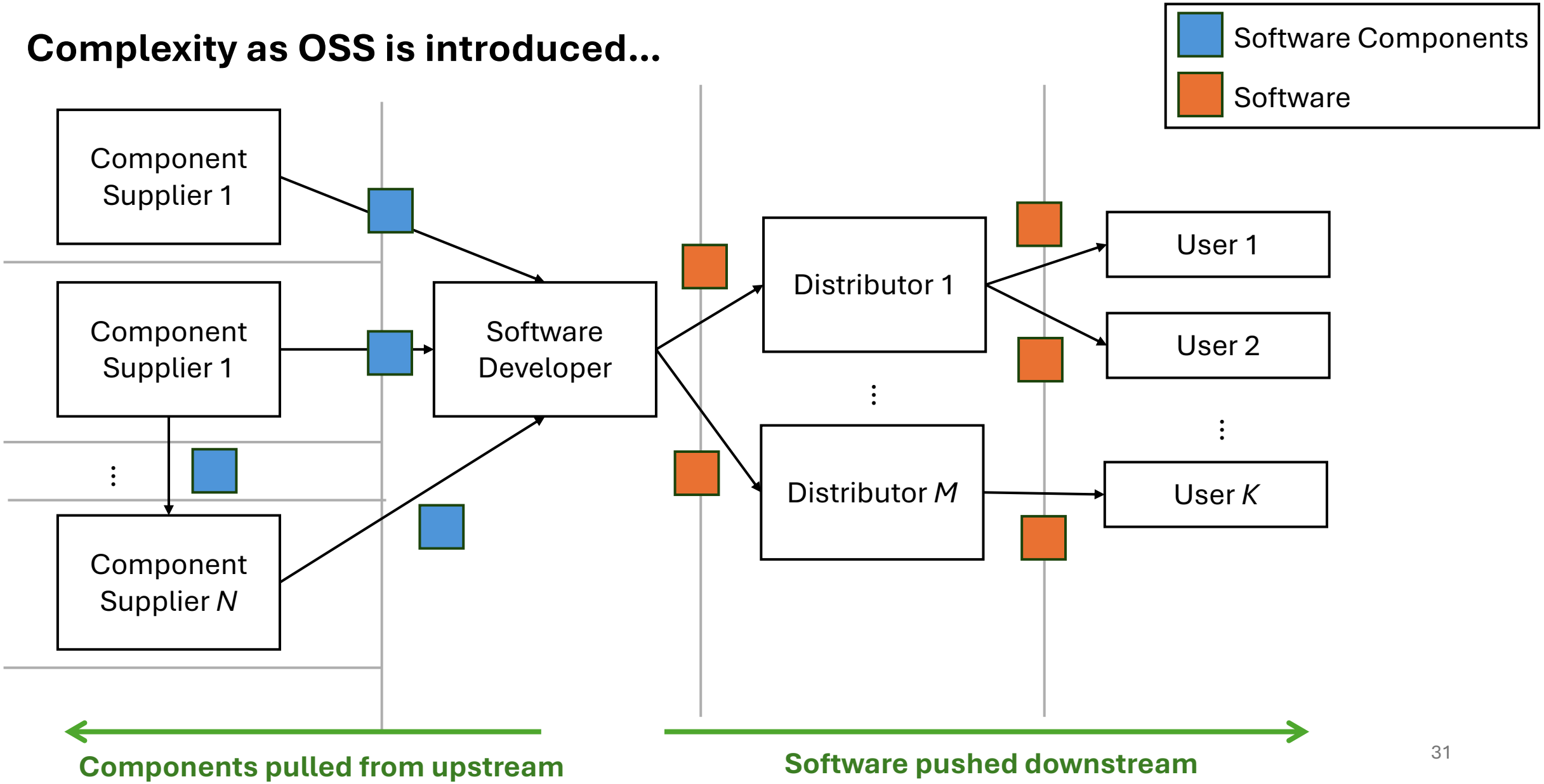
- Package & push latest version to the distribution platform

Deployment Phase



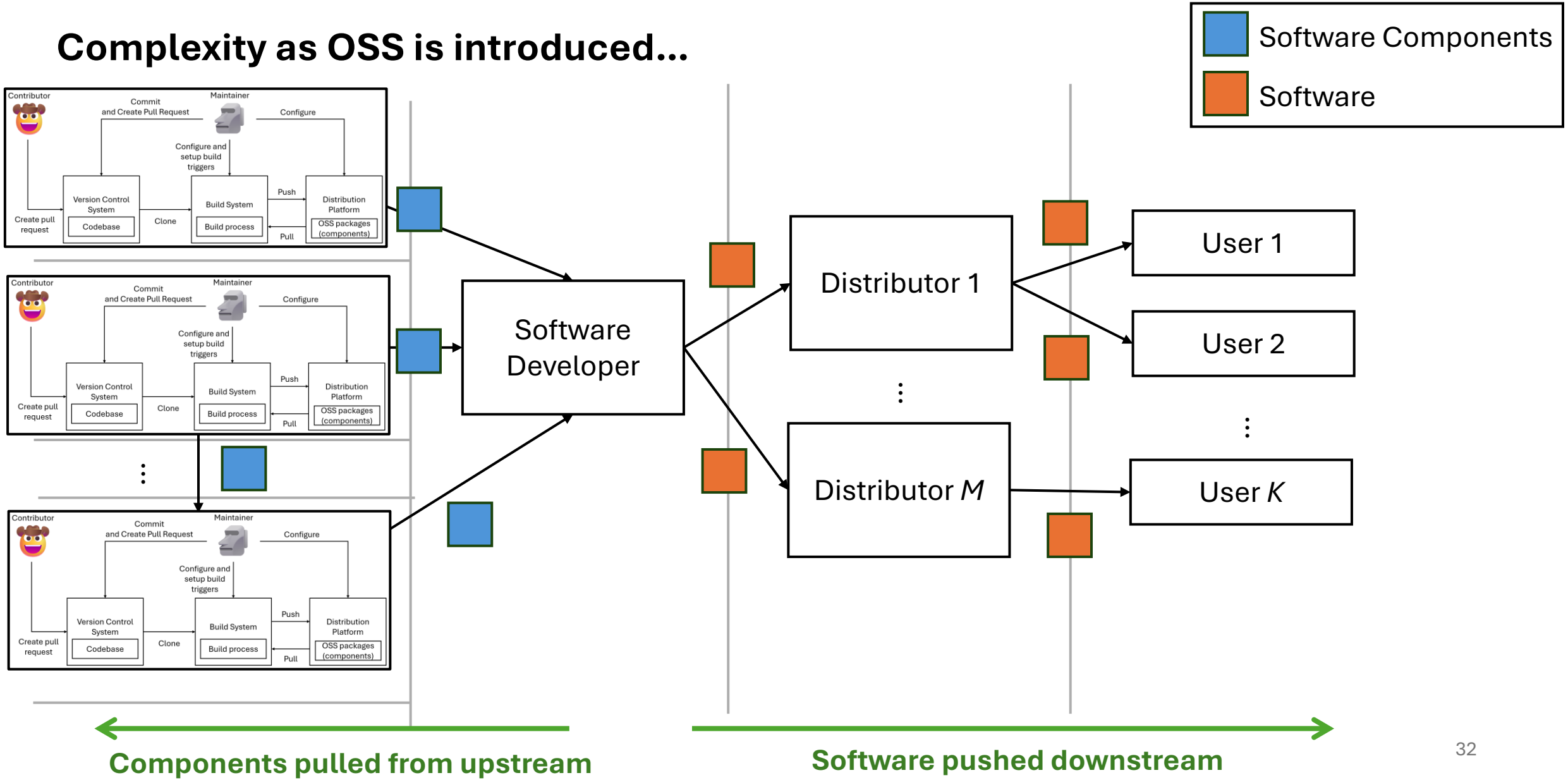
Software Supply Chain Models

Complexity as OSS is introduced…



Software Supply Chain Models

Complexity as OSS is introduced…



Software Supply Chain Attacks

Recall from "*Reflections on Trusting Trust*"

- *"You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me)."*

How many attack surfaces are there within SBOM?

- Malicious OSS contributions → malware
- Malicious maintainers → backdoor in build system, distribution platform
- Malicious dependencies
- Malicious software developers
- Malicious distributors / manufacturers
- Too many to count!

Software Supply Chain Attacks

Can become quite complex and overwhelming depending on where you look…

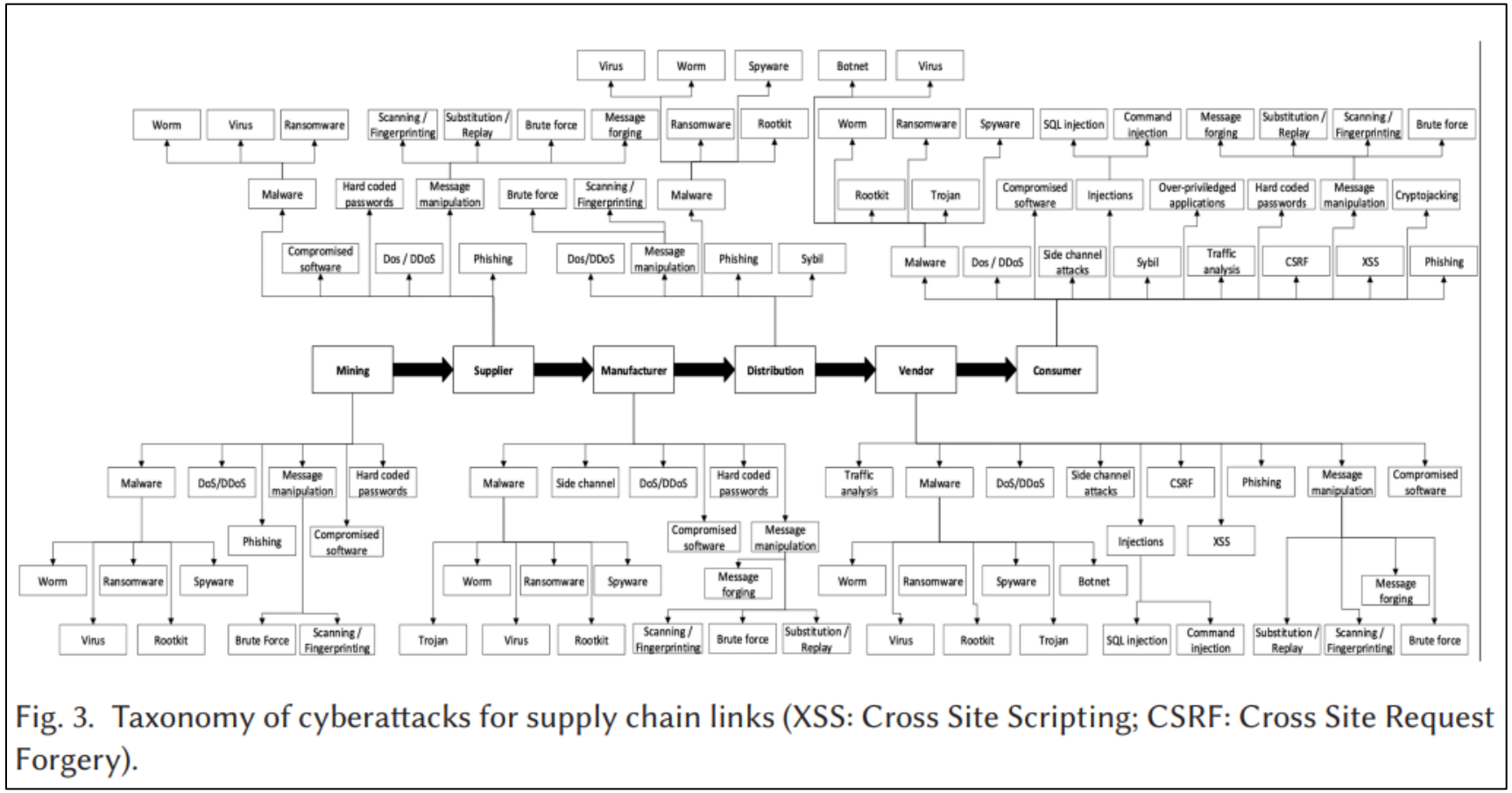


Fig. 3. Taxonomy of cyberattacks for supply chain links (XSS: Cross Site Scripting; CSRF: Cross Site Request Forgery).

Figure from "*Security Threats, Countermeasures, and Challenges of Digital Supply Chains*"

Software Supply Chain Attacks

Can become quite complex and overwhelming depending on where you look…

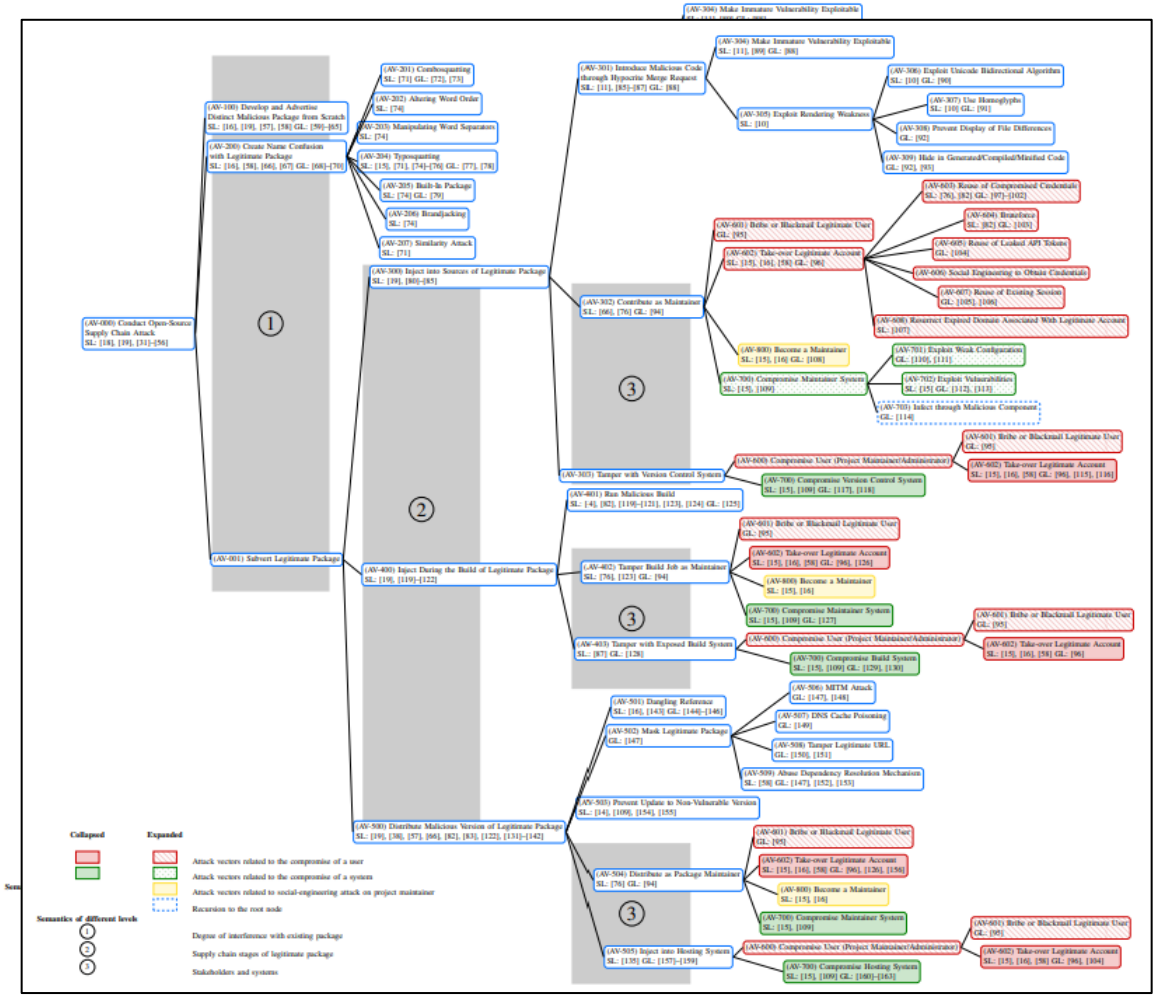


Figure from “*SoK: Taxonomy of Attacks on Open-Source Software Supply Chains*”

Software Supply Chain Attacks

Let's fine-tune the focus…

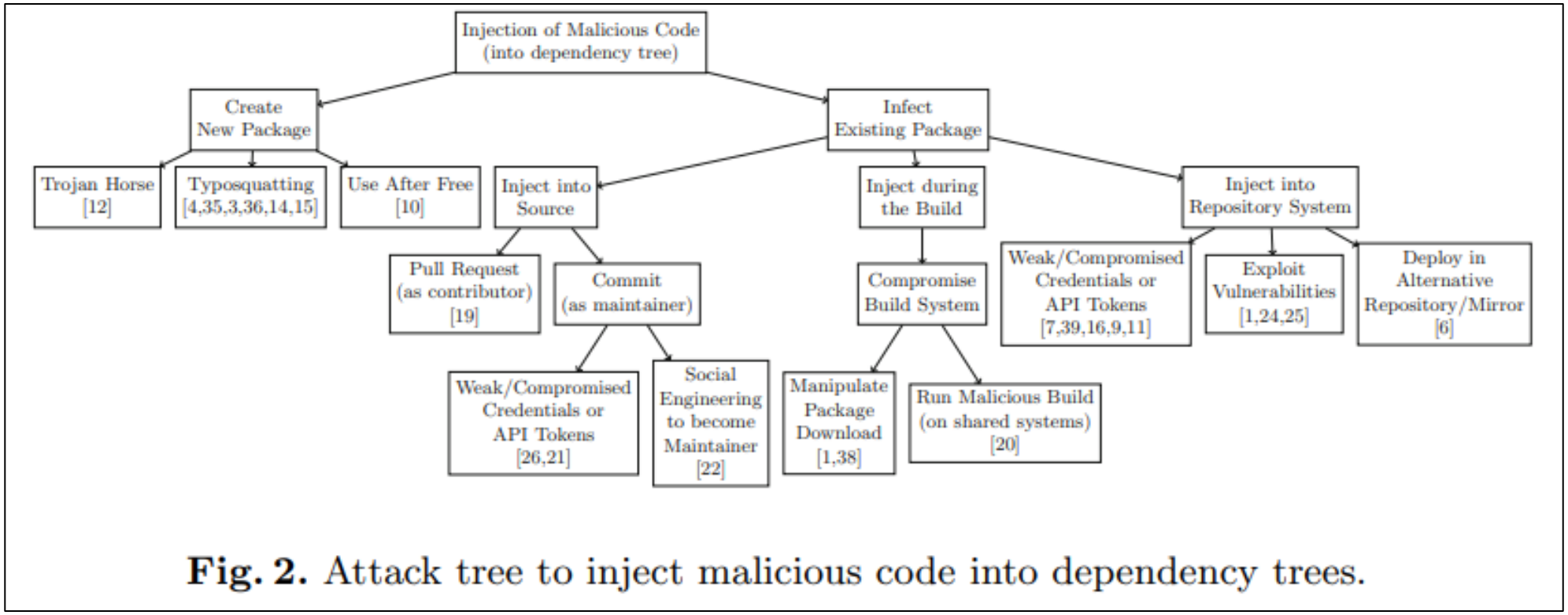


Fig. 2. Attack tree to inject malicious code into dependency trees.

Figure from "*Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks*"

Software Supply Chain Attacks

Classes of attacks and examples

Injection of code

- Create a new *malicious* package
- Avoids interference with other legitimate project maintainers
- Challenge → has to be discovered and referenced by downstream users
- Trojan horses
- *Use (package/project/account name) after free*
- Prey on human error → *typosquatting*

Software Supply Chain Attacks

Typosquatting: take advantage of user typos/mistakes to cause install of malicious package from distribution platform

1. Create a new package with a name similar to a popular package that includes the malicious code in the new package. Examples:

- Squat on PyPI the Debian package name: *python-sqlite* vs. *sqlite*
- English variants (color vs. colour)
- Unicode tricks → requests vs. requests

2. Upload it to a distribution platform (e.g., PyPI)

3. Wait for users to mistype (e.g.,. *pip install python-sqlite*)

Software Supply Chain Attacks

Classes of attacks and examples

Infect existing package

- Adversary may target an influential package
  - Many downstream users
  - Specific downstream user group
- Inject into sources, during the build, or into package repositories
- Malicious pull request: claims to fix a bug or add a useful feature
- Direct commits using weak or compromised credentials
- Social engineering to become a maintainer
- Compromised build system → e.g., compilers, network services
  - Manipulate package downloads during build

Software Supply Chain Attacks

Classes of attacks and examples

Inject into repository system

- Use weak or compromised credentials
- Gain maintainer authorizations through social engineering
- Malicious package versions to alternative repositories or repository mirrors
- Less common attack vector

Software Supply Chain Attacks

Some notable examples:

CCleaner malware infection in 2017

- Software for managing apps and system maintenance (cleaning files)
- Attack stage: build / artifact creation
- Internal compromise: inserted malware into a software update
- Compromised before signing: user's received a signed update

SolarWinds attack 2019-2020

- Malicious update in SolarWinds Orion monitoring software
- A hidden file injected among the update

XZ Utils backdoor 2024

- Backdoor added to Linux build of the xz compression library
- Malicious maintainer → obtained through social engineering

Software Supply Chain Defenses

Possible safeguards for each entity along the supply chain

- Safeguards for consumers
- Safeguards for maintainers

Software Supply Chain Defenses

Safeguards for Consumers

- Can opt to build packages directly from source rather than pre-built artifacts
- Eliminate all risks related to the compromise of third party build services and repos
- Isolating the code
- Sandboxing it during execution
- Establishment of internal repository mirrors

Software Supply Chain Defenses

Safeguards for Maintainers

- Secure authentication
- Multi-factor authentication and strong password policies
- Maintain SBOM and perform dependency analysis
- Use hosted publicly accessible VCSs
  - Careful merge request reviews
  - Enable branch protection rules for sensitive branches
- Dedicated build services
  - Ephemeral environments
  - Isolated build steps – how?
- Reproducible builds
- Integrity checks on dependencies & SBOM

Reproducible Builds

Definition: *the build process of a software product is reproducible if, after designating a specific version of its source code and all of its build dependencies, every build produces bit-for-bit identical artifacts, no matter the environment in which the build is being performed.*

Threat model:

- Untrusted – the build process completed by a third party
- Trusted – multiple independent third parties (e.g., other users)

Users can establish trust in built executables if

- Developers submit a checksum of an artifact along with the source code
- User pulls source code, produces artifact, takes checksum
- Compare artifacts

Reproducible Builds

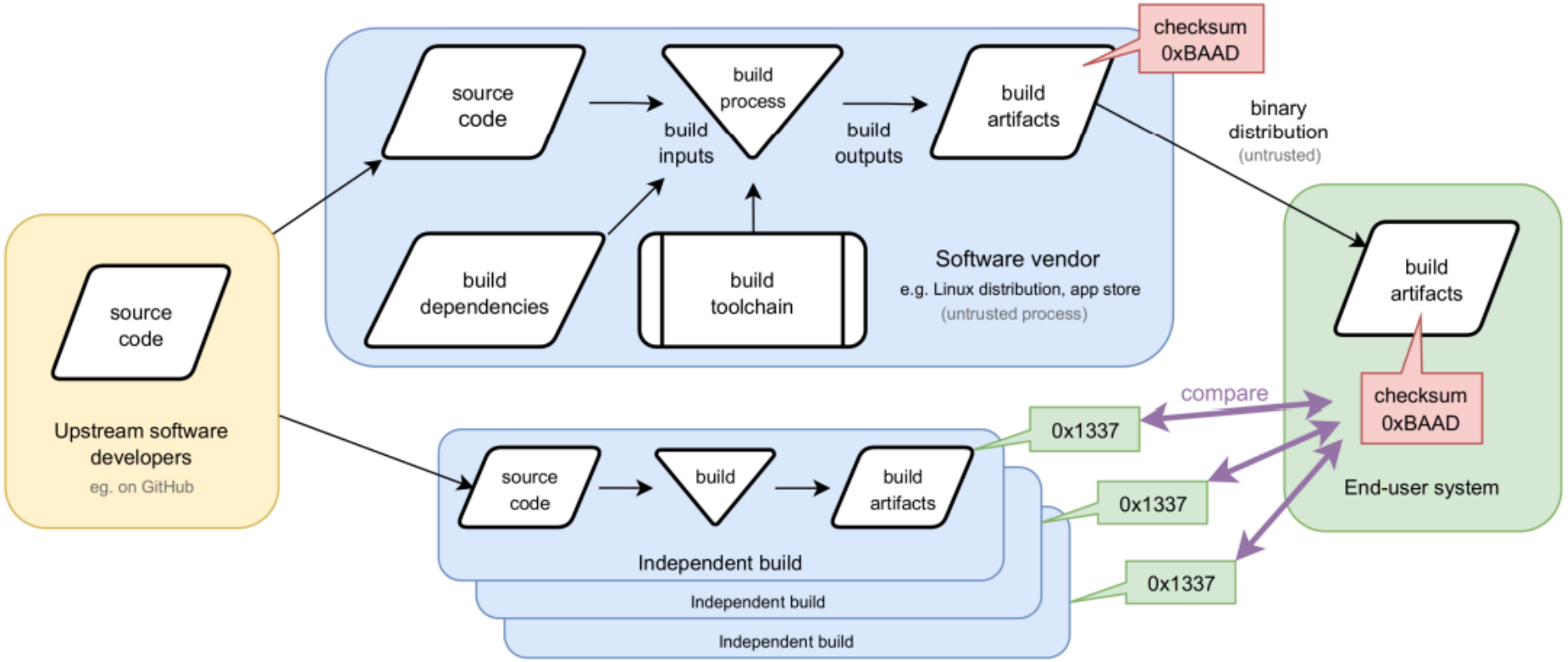


Figure 1. The *reproducible builds* approach to increasing trust in executables built by untrusted third parties. The end-user should reject the binary artifact from their software vendor, as its checksum (`0xBAAD`) does not match the one built by multiple, independent third-parties (`0x1337`).

Figure from: "*Reproducible Builds: Increasing the Integrity of Software Supply Chains*"

Reproducible Builds

Challenges: can we make bit-for-bit identical artifacts?

- Multiple inputs → sources, linker, compiler, etc.
- Multiple outputs → executables, data, documentation, etc.
- Build still may be unreproducible
  - Due to *uncontrolled build inputs* and *build non-determinism*

Uncontrolled build inputs

- Occur when toolchains allow the build process to be affected by the environment
- System time, environmental variables, arbitrary file paths

Build non-determinism

- Occurs when aspects of the build have non-deterministic characteristics or behaviors
- Some part of the output depends on a PRNG

Reproducible Builds

How to face these challenges?

One idea… could build a sanitized environment

- Always presents the same interface to the build system – “jails”
- Impose technical restrictions
- Slower build times
- Doesn’t address non-determinism

Must ensure that:

- Build depends on only legitimate build inputs:
  - Source code, build dependencies, and the toolchain
- Non-deterministic behaviors do not affect the resulting artifacts

Reproducible Builds

Some other sources of non-determinism

Timestamps

- One of the biggest sources of irreproducibility
- Common practice to embed dates into binaries (e.g., `__DATE__` macro in C)
- Also within archive metadata

Build paths & filesystem ordering

- The build path itself is sometimes embedded into the generated binaries
- Ordering and access may be random: linking of executables in arbitrary order

Randomness

- Builds that iterate over hash tables: elements are returned in arbitrary order
- Parallelism with arbitrary completion
- Uninitialized memory

Reproducible Builds

Tools – reproducible-builds.org

Case study: in-toto framework

in-toto framework

- Cryptographic approach
- Grants the end user the ability to verify the software supply chain
- From the inception (software components / artifacts) to the final product

Case study: in-toto framework

Security Goals of in-toto

- Supply Chain Layout integrity
  - All steps defined in a supply chain are performed in the specified order
- Artifact flow integrity
  - All artifacts created or transformed by a step must not be altered in-between
- Step authentication
  - Steps can only be performed by intended parties
  - No party can perform a step unless it is given explicit permission to do so
- Implementation transparency
  - Existing supply chains should not be required to change their practices to implement in-toto

Case study: in-toto framework

Step 1: Constructing a Layout

Specifies all the valid steps, actors, and other conditions for the build:

- “Steps”
- A readme to provide helpful description
- Public keys of all of the expected actors along the supply chain
- Inputs and outputs of each “step”
- A list of inspections aka verification steps after product delivery
- Layout descriptive language
- Signed by the Project Owner

Case study: in-toto framework

Step 1: Constructing a Layout -- Example

```
1{  "_type" : "layout",
2   "expires" : "<EXPIRES>",
3   "readme": "foo.pkg supply chain",
4   "keys" : {"<BOBS_KEYID>":"<PUBKEY>",
5             "<ALICES_KEYID":"<PUBKEY>",
6             "<CLARAS_KEYID":"<PUBKEY>"},
7   "steps" : [{"name": "tag", "..."},
8              {"name": "build", "..."},
9              {"name": "package", "..."} ],
10  "inspections" : ["{"name": "inspect", "..."}]
11}
```

Listing 2: The supply chain for our example

Example from in-toto paper

Case study: in-toto framework

What is a “step” in in-toto?

Describes the actions that should be taken by a particular actor (and other metadata):

- Name → unique identified that describes the step
- Expected materials → expected input artifacts
- Expected products → expected output artifacts
- Expected command → the expected command to execute (and flags)
- Threshold → minimum pieces of signed link metadata that must be provided for verification
- A list of public key ids → IDs of the keys that can be used to sign the link metadata for this step

Case study: in-toto framework

What is a “step” in in-toto?

```
{  "_name": "<NAME>",
   "threshold": "<THRESHOLD>",
   "expected_materials": [["<ARTIFACT_RULE>"], "..."],
   "expected_products": [["<ARTIFACT_RULE>"], "..."],
   "pubkeys": ["<KEYID>", "..." ],
   "expected_command": "<COMMAND>"
}
```

Listing 3: A supply chain step in the supply chain layout

```
{  "_name": "build",
   "threshold": "1",
   "expected_materials": [
     ["MATCH", "foo.c", "WITH",
      "PRODUCTS", "FROM", "tag"]
   ],
   "expected_products": [["CREATE", "foo"]],
   "pubkeys": ["<BOBS_PUBKEY>"],
   "expected_command": "gcc foo.c -o foo"
}
```

Listing 4: The build step in our example layout

Examples from [in-toto paper]

Case study: in-toto framework

Step 2: Produce updated Link metadata files

- Serves as a record that the steps in the layout took place
- Includes measurements & metadata of the steps
- Name → used to identify the corresponding step
- Materials → input files & their cryptographic hashes
- Command → the command run along with its arguments
- Products → outputs produced and their cryptographic hashes
- Byproducts → reported information about the steps (error buffers, stdout)
- Signature → a cryptographic signature over the Link

Case study: in-toto framework

What is a "step" in in-toto?

```
{ "_type" : "link",
  "_name" :  "<NAME>",
  "command" : "<COMMAND>",
  "materials": {"<PATH>": "<HASH>", "..." : "..." },
  "products": {"<PATH>": "<HASH>", "..." : "..." },
  "byproducts": { "stdin": "", "stdout": "",
        "return-value": "" },
  "environment": {"variables": "<ENV>",
        "filesystem": "<FS>",  ...}
}
```

Listing 7: Link metadata format

```
{ "_type": "link",
  "name": "build",
  "command": ["gcc", "foo.c", "-o", "foo" ],
  "materials": {"foo.c": { "sha256": "bff95e..."}},
  "products": {"foo": {"sha256": "25c696..."}}
  "byproducts": { "return-value": 0,
        "stderr": "", "stdout": ""},
  "environment": {},
}
```

Listing 8: The link for the build step

Examples from in-toto paper

Case study: in-toto framework

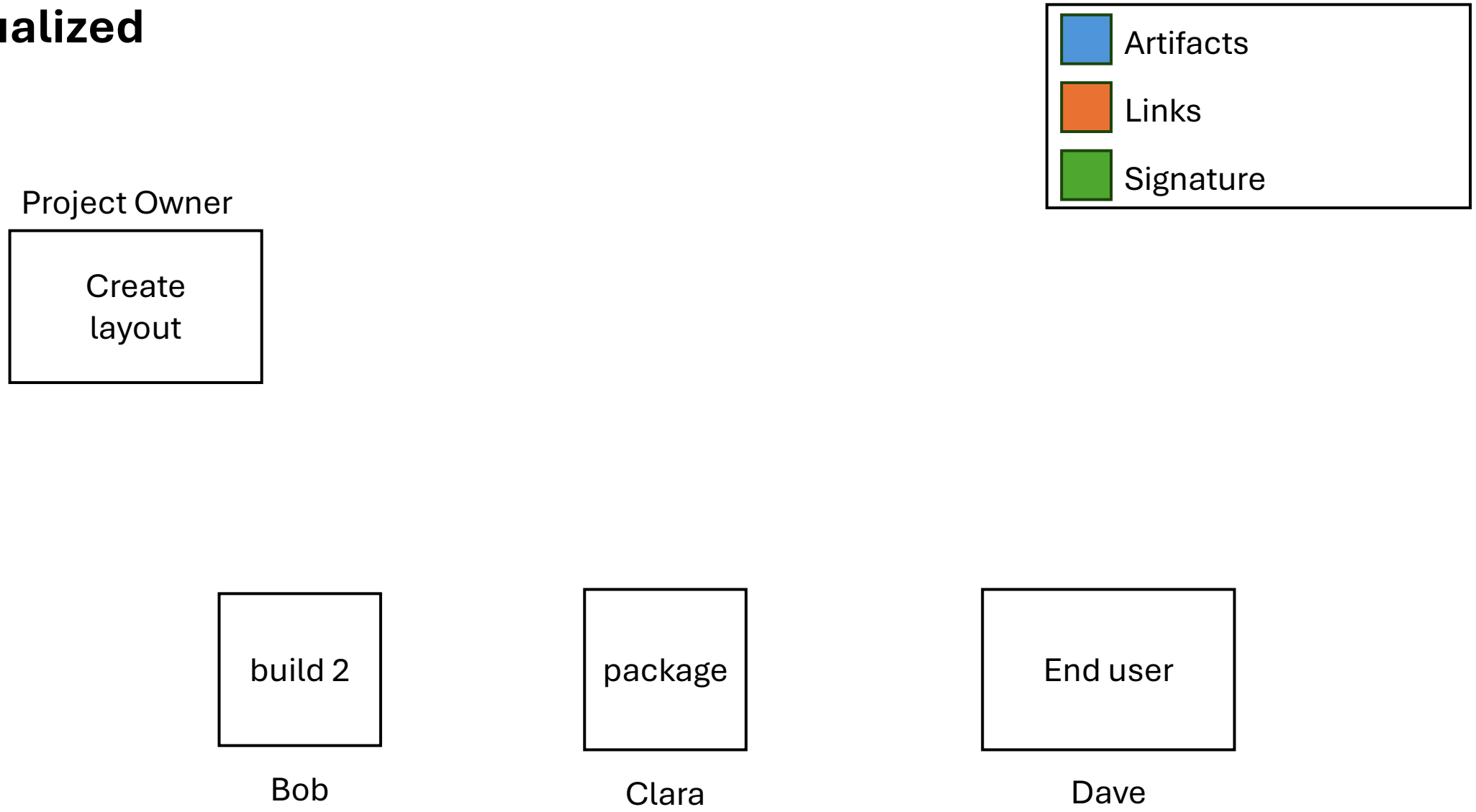
Step 3: Verification

- The end user obtains
  - The result of the supply chain sequence (e.g., resulting artifacts)
  - The initial layout file (containing all the public keys)
  - The sequence of signed Link metadata files describing the steps

- Verify the sequence to determine if
  - All steps were performed by valid actors
  - If they produced the expected outputs at each step

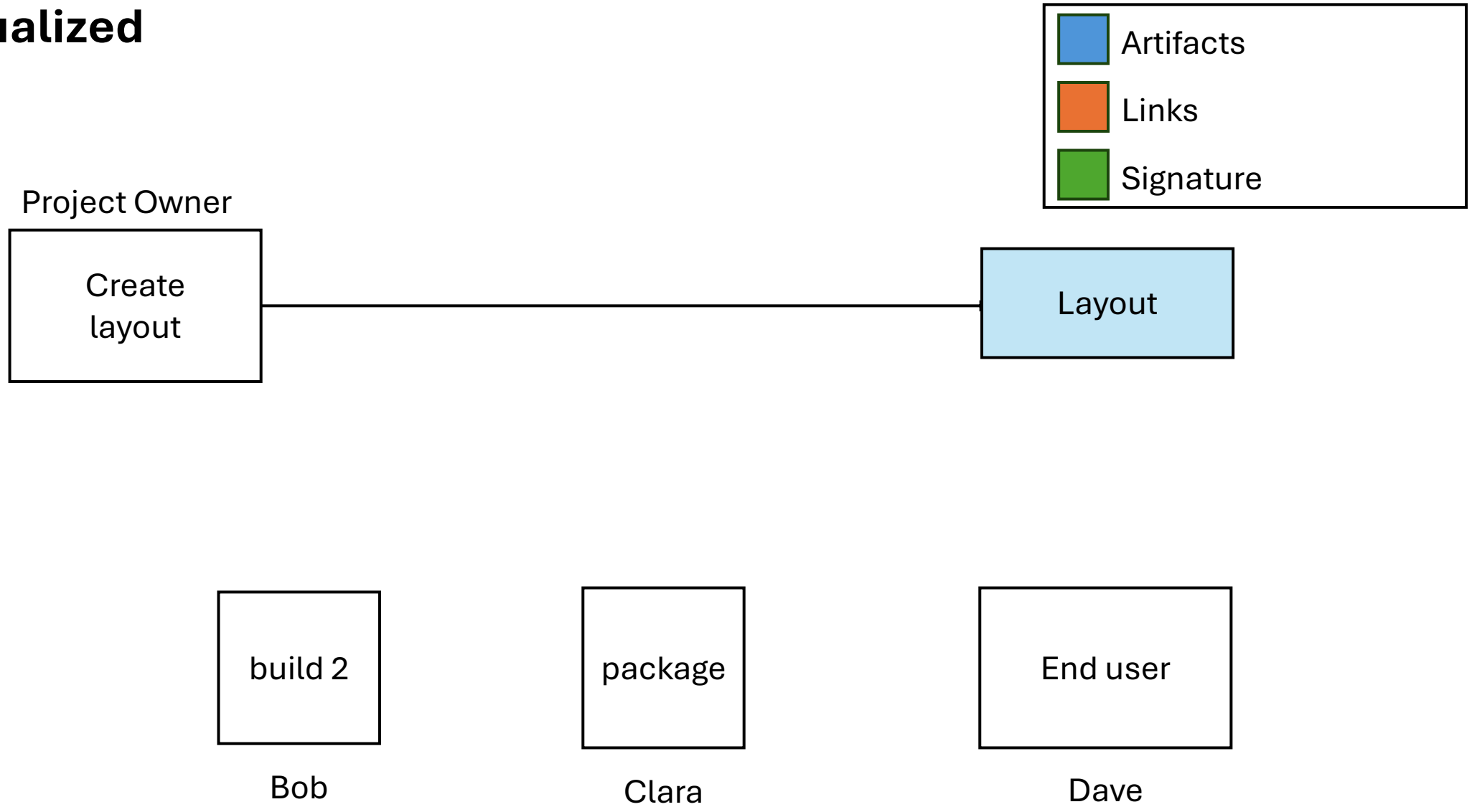
Case study: in-toto framework

in-toto visualized



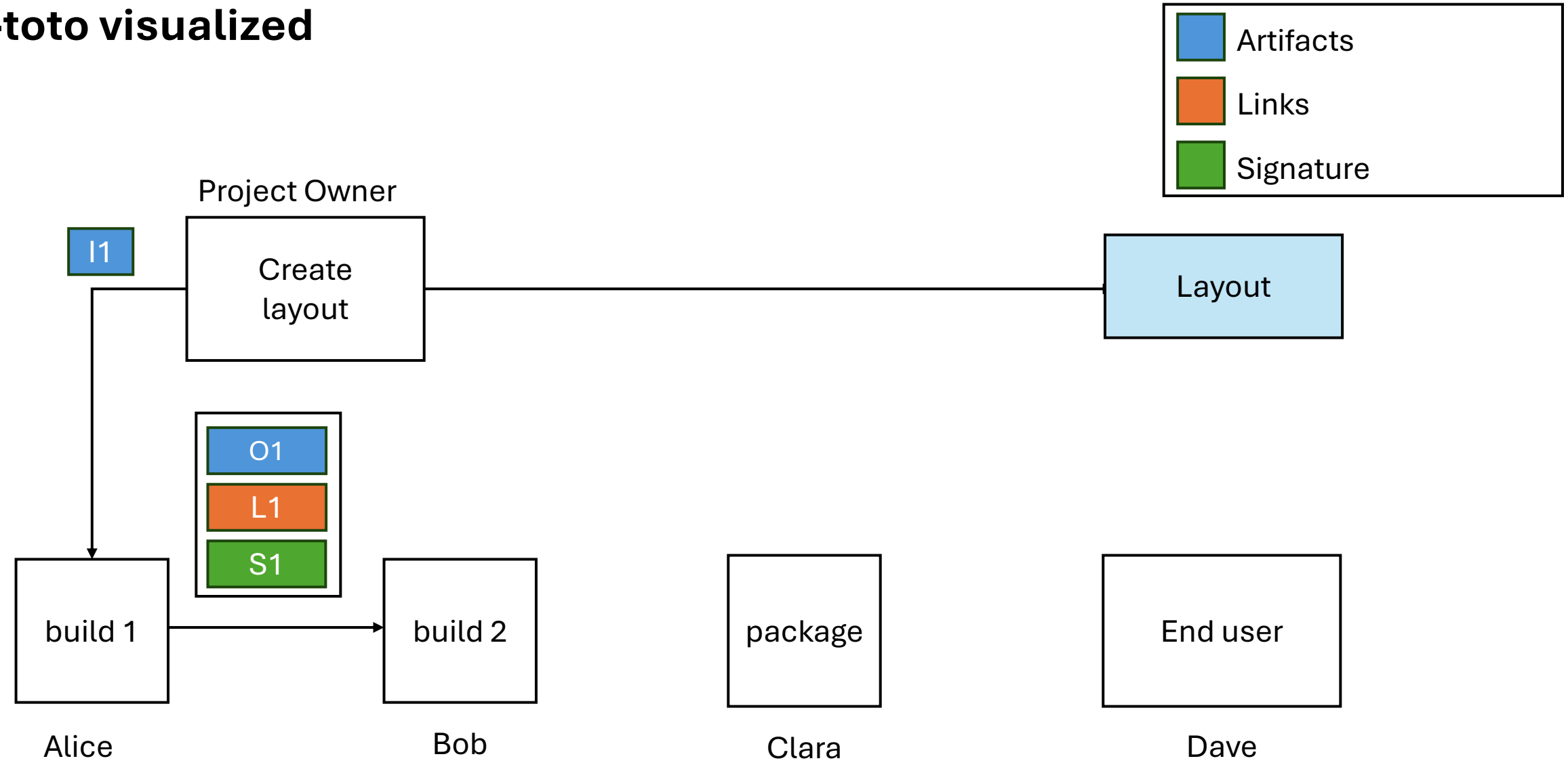
Case study: in-toto framework

in-toto visualized



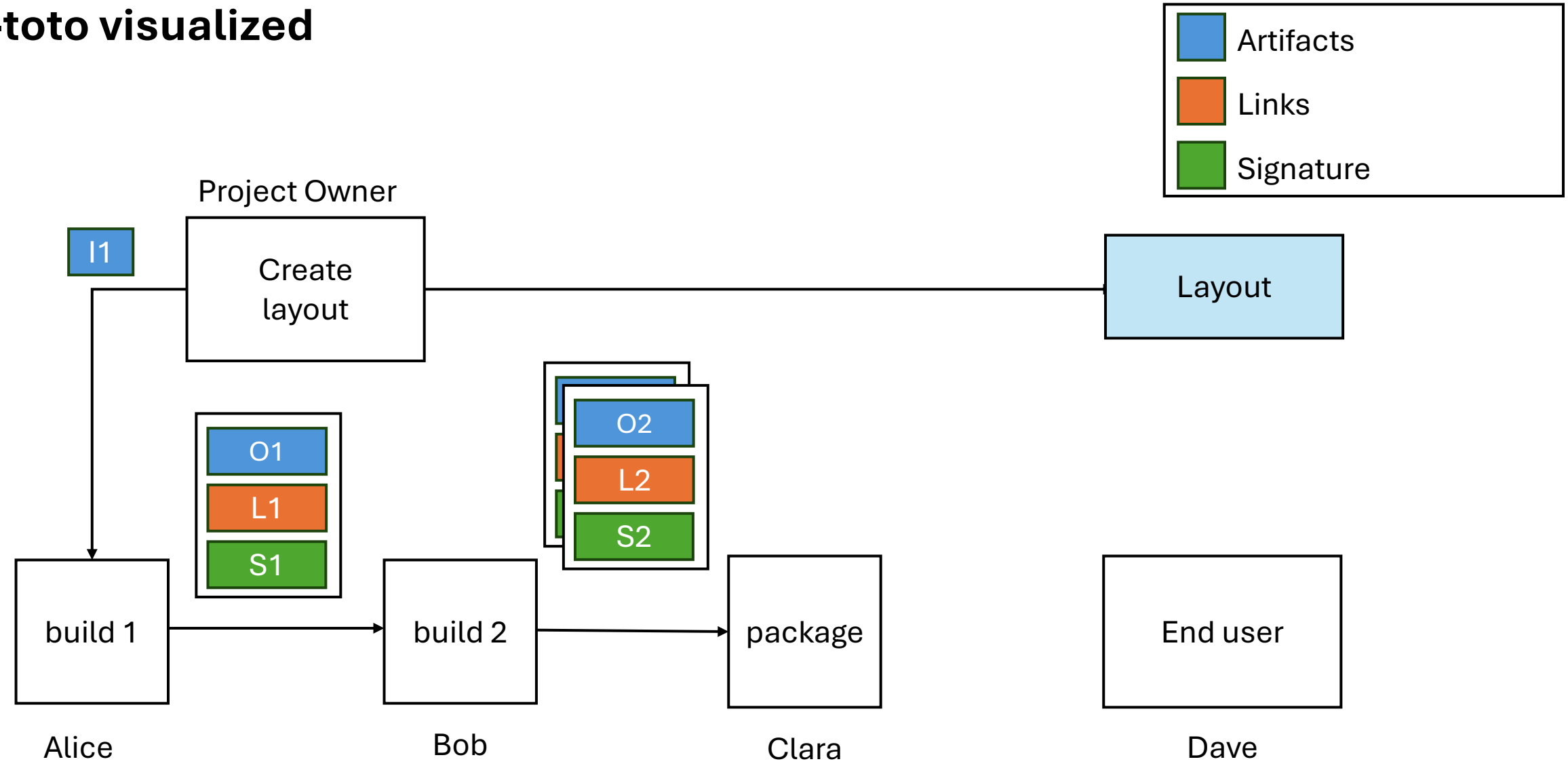
Case study: in-toto framework

in-toto visualized



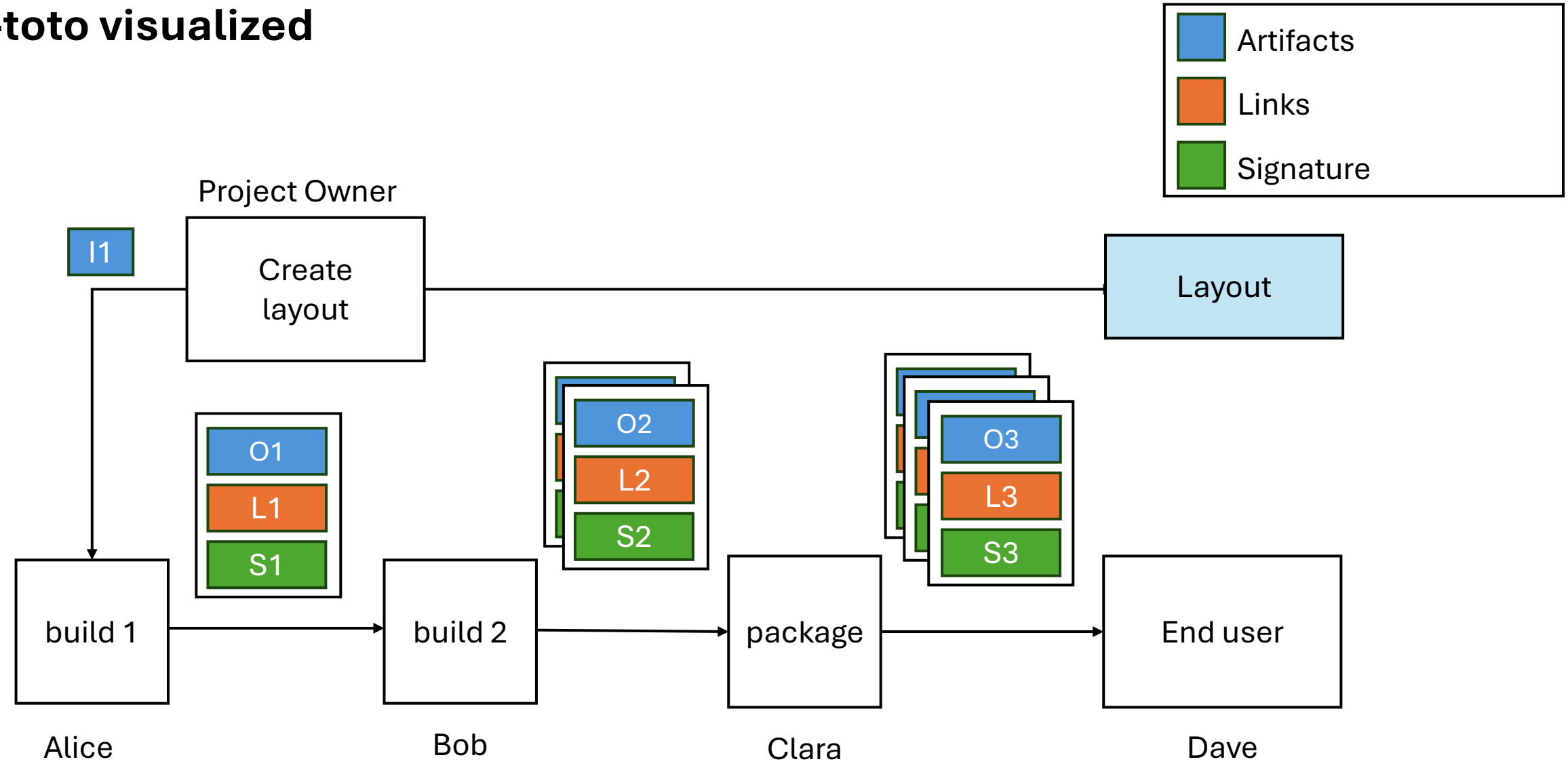
Case study: in-toto framework

in-toto visualized



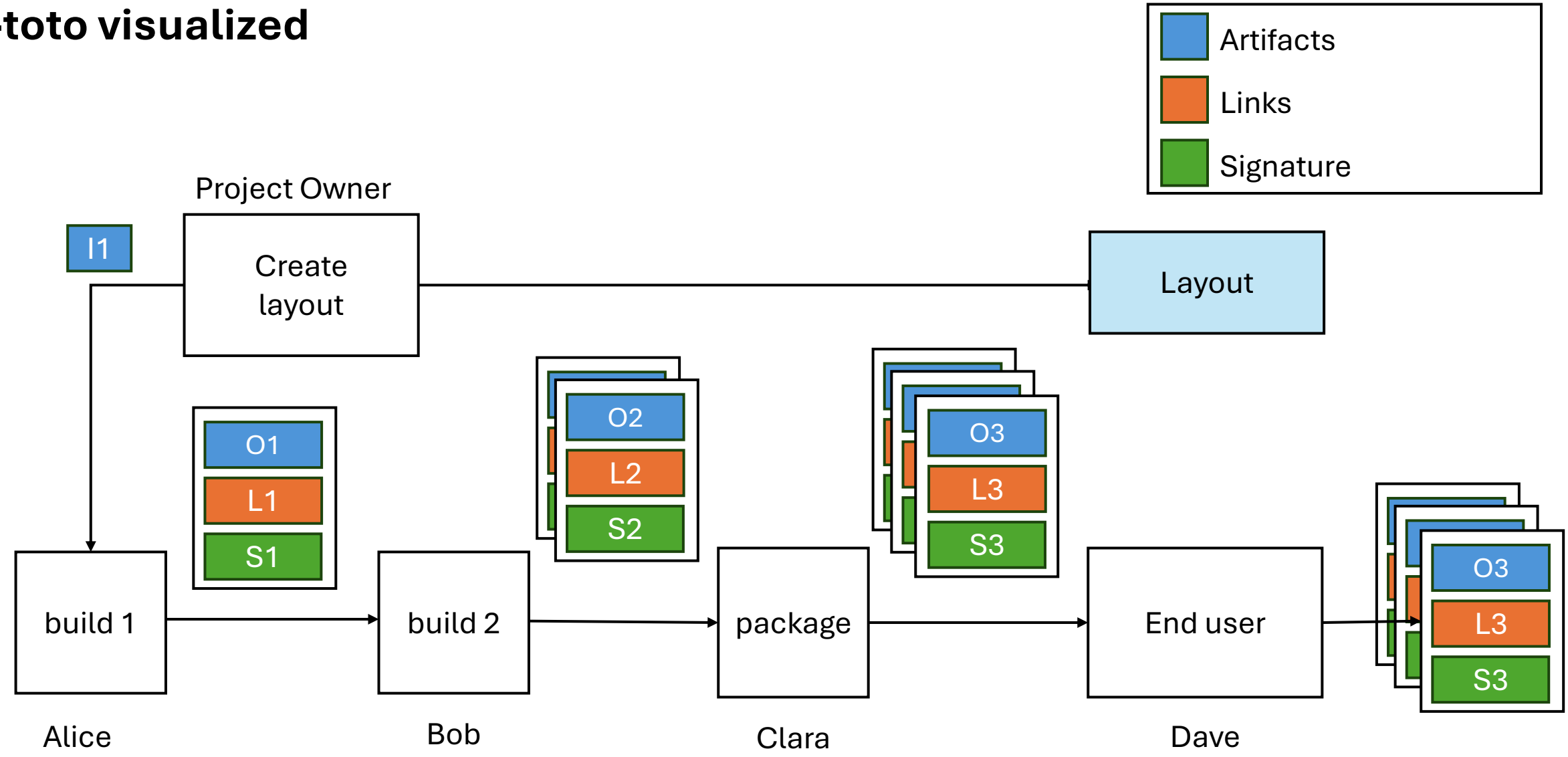
Case study: in-toto framework

in-toto visualized



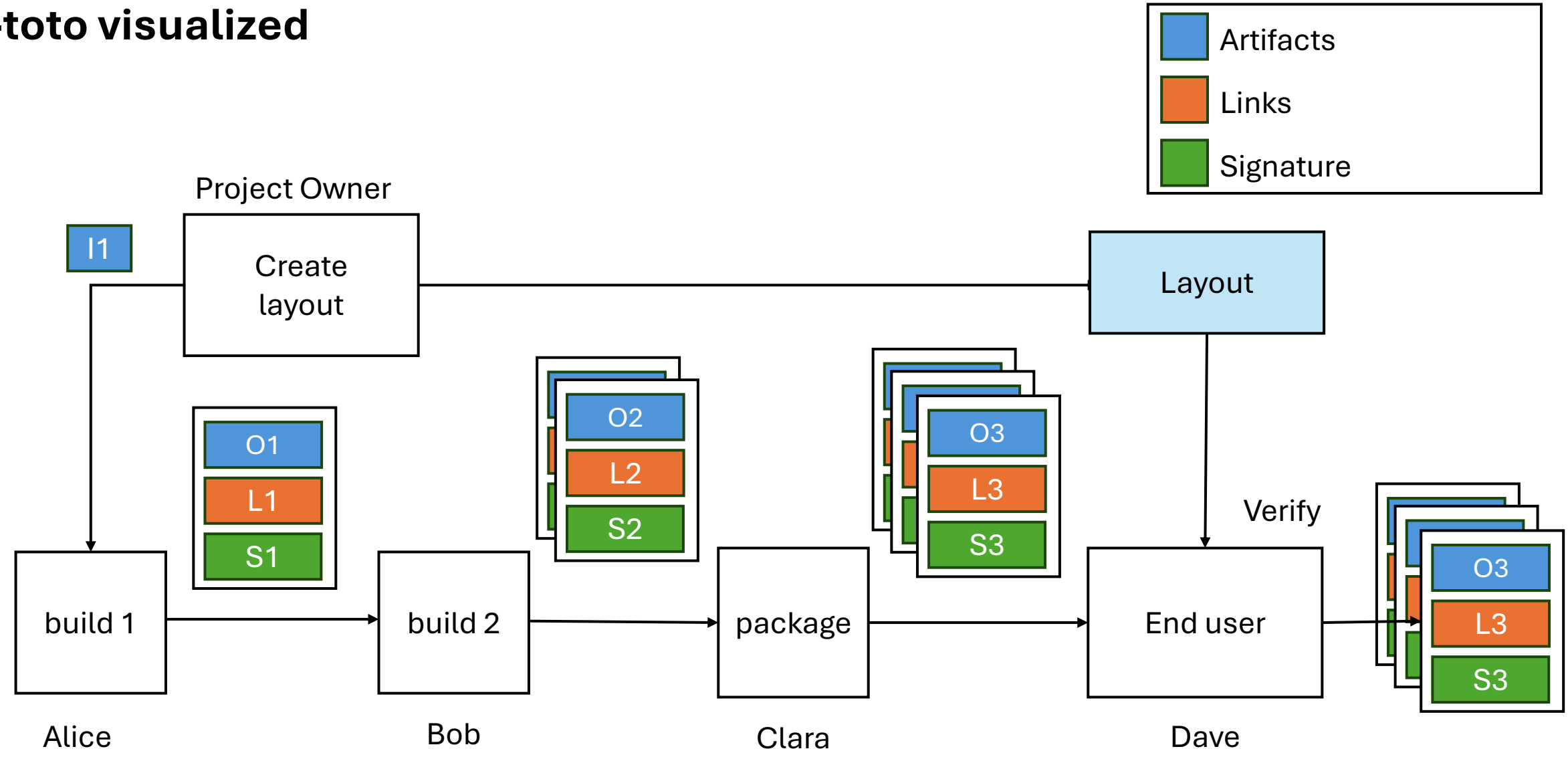
Case study: in-toto framework

in-toto visualized



Case study: in-toto framework

in-toto visualized



Case study: in-toto framework

Concluding questions and thoughts…

Is in-toto a form of *authentication or attestation*?

Security of in-toto depends on some key assumptions from the paper:

- *When an attacker is able to compromise infrastructure or communication channels **but not keys**, in-toto's security properties ensure integrity is upheld*
- *It is important to underline that this threat model requires that the host system is not compromised*

How can we obtain these guarantees? → coming up…

That’s all for today!

Resources from today….

- Supply chain security: definitions, theory, models
  - ”On Systematics of the Information Security of Software Supply Chains”
  - “Supply Chain Attacks and Resiliency Mitigations”
  - “Research Directions in Software Supply Chain Security”
  - “SoK: Taxonomy of Attacks on Open-Source Software Supply Chains”
  - “Backstabber’s Knife Collection: A Review of Open Source Software Supply Chain Attacks”
  - “Security Threats, Countermeasures, and Challenges of Digital Supply Chains”
  - “Software Supply Chain Attacks An Illustrated Typological Review”
- Reproducible builds
  - Paper
  - Tools
- in-toto framework
  - USENIX Security Paper
  - Video presentation
  - Website + GitHub with demo

That’s all for today!

Coming up….

- What can be used to obtain roots of trust (RoT) at run-time?
  - Secure boot?
  - Something else? (… here comes Hardware & Mobile Security …)

Reminders:

- A3 is due on July 11
- Research project proposals