TypeJinja: Static Type Checking of Jinja Templates at dbt Labs

Cheng Ding
The University of Texas at Austin
United States
cheng.ding@utexas.edu

Zhong Xu
dbt Labs
United States
zhong.xu@dbtlabs.com

Michael Y. Levin
dbt Labs
United States
michael.levin@dbtlabs.com

Wolfram Schulte
dbt Labs
United States
wolfram.schulte@dbtlabs.com

Milos Gligoric
The University of Texas at Austin
United States
gligoric@utexas.edu

Abstract

Jinja is a templating language widely used in many domains, including web development and data engineering. At dbt Labs, we extensively use Jinja to enable dynamic SQL rendering via the dbt fusion engine, a popular data transformation tool. However the flexibility of Jinja templates (i.e., lack of types) can lead to errors in the rendered SQL code, which can be costly to debug and fix. To address this issue, we present the design and implementation of TypeJinja: a typed version of the Jinja language and its integration with the dbt fusion engine. TypeJinja statically analyzes Jinja templates and their context to detect type errors, before rendered SQL code ends up being used, providing developers with early feedback. Furthermore, TypeJinja is designed to be extensible, allowing it to be adapted to different use cases. We have evaluated TypeJinja on our codebase and found 30 previously unknown type errors, demonstrating its effectiveness in improving code quality and reducing debugging effort. TypeJinja is now part of the publicly available GitHub repository that hosts the dbt fusion engine.

CCS Concepts

• **Software and its engineering → Domain specific languages**; **Data types and structures**; • **Information systems → Structured Query Language**.

Keywords

Jinja, Program Analysis, Type Checking, Data Transformation

ACM Reference Format:
Cheng Ding, Zhong Xu, Michael Y. Levin, Wolfram Schulte, and Milos Gligoric. 2026. TypeJinja: Static Type Checking of Jinja Templates at dbt Labs. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-SEIP '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3786583.3786905

1 Introduction

Jinja [36] is a widely used templating language that allows developers to embed programmatic constructs into text-based documents. Originally used for web development, it has since become a general-purpose templating engine employed across a wide range of domains [28, 34, 38].

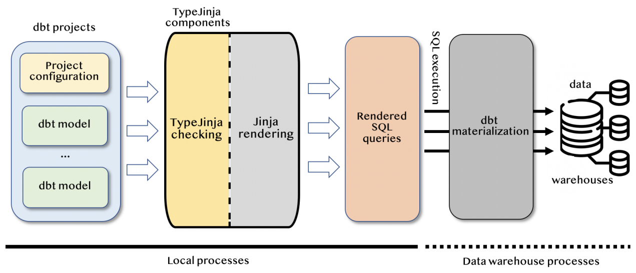


Figure 1: The workflow of dbt. TypeJinja catches type errors before models even go through rendering.

At dbt Labs [10], where we focus on the data engineering ecosystem, we heavily use Jinja for writing data transformation pipelines [6]. We maintain a fork of MiniJinja [23], which is a Rust-based reimplementation of the popular Jinja templating engine. As shown in Figure 1, a user writes templates (known as *models* in our domain) that MiniJinja renders into SQL queries. MiniJinja enables users to easily compose modular query fragments and conditional logic that adapts to various SQL dialects (e.g., Snowflake, BigQuery).

By blending Python-like syntax (from Jinja) with SQL, dbt empowers developers to express complex data transformations in a more maintainable and reusable fashion.

However, this flexibility comes at a cost. Because MiniJinja is dynamically typed (like Python and JavaScript), errors in models' logic often remain latent until the rendered SQL is executed. The consequences of these errors can be significant, because SQL code executes against production databases and can perform unexpected transformations. Mistakes in type usage, such as passing a string where a numeric value is expected or invoking a macro with incorrect argument types, can lead to subtle bugs that are difficult to detect. In practice, many such issues escape into production, because only extreme levels of exhaustive testing for dynamically typed languages could match the power of static type checking for discovering type errors. Writing exhaustive tests for all possible model inputs is infeasible, and failures may only appear in some specific inputs. Engineers at dbt Labs have observed that these problems lead to significant maintenance overhead, debugging effort, and in some cases silent incorrectness of rendered SQL queries.

To address this problem, we introduce TypeJinja: the design and implementation of a typed version of MiniJinja language used at

```
{# (list[string]) -> string #}
{% macro generate_payment_amounts(payment_methods) %}
select
    order_id,
    {% for payment_method in payment_methods %}
    sum(case when payment_method = '{{payment_method}}' then amount end) as {{payment_method}}_amount,
    {% endfor %}
    sum(amount) as total_amount
from app_data.payments
group by 1
{% endmacro %}
```

(a) Jinja model.

```
select
    order_id,
    sum(case when payment_method = 'bank_transfer' then amount end) as bank_transfer_amount,
    sum(case when payment_method = 'credit_card' then amount end) as credit_card_amount,
    sum(case when payment_method = 'gift_card' then amount end) as gift_card_amount,
    sum(amount) as total_amount
from app_data.payments
group by 1
```

(b) Rendered SQL.

Figure 2: Comparison of a Jinja model (a) and its rendered SQL (b), shown in dbt's official documentation [9].

dbt Labs. We implemented TypeJinja and integrated it into the dbt workflow, as shown in Figure 1. Our approach integrates a type-checking phase directly into the MiniJinja compilation phase. After a model is parsed into an Abstract Syntax Tree (AST) and translated into a MiniJinja intermediate representation (IR), TypeJinja analyzes the IR before IR instructions are rendered into SQL. By reasoning about the types of variables, macro parameters, and expressions at the IR instruction level, we can identify errors without rendering the models or requiring concrete inputs for models. Our analysis proceeds through a flow-sensitive data-flow framework, checking that each IR instruction is consistent with the type system we define.

This allows dbt users to catch mismatches such as undefined variables, invalid macro calls, or unsafe attribute accesses early in the development.

Applying TypeJinja to a large collection of MiniJinja models, drawn from dbt libraries which contain a large number of reusable macros discovered a substantial number of unknown bugs in those models. Specifically, we analyzed a project containing 178 models comprising 6,869 lines of code, and TypeJinja uncovered 30 distinct type errors. Many of these errors were subtle and would have been extremely difficult to identify using tests alone, such as macros invoked with missing arguments or variables that are only defined along some control-flow branches but then used in others where they may be undefined.

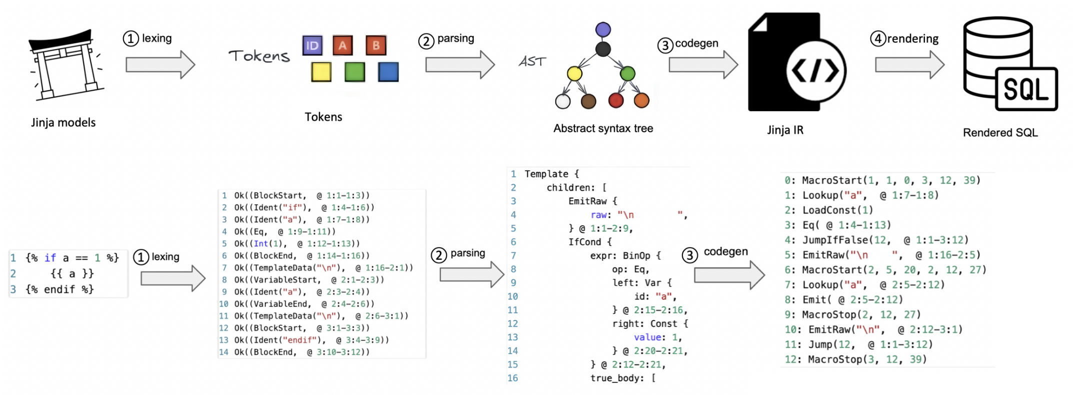
As expected, our results demonstrate that static type checking provides a powerful complement to existing testing strategies, substantially improving the reliability of MiniJinja models in production settings.

We also aimed to improve usability and extensibility of TypeJinja. One challenge with applying type systems to templating languages is that dbt employs domain-specific abstractions that are not captured by a fixed set of built-in types. To accommodate this diversity, TypeJinja allows dbt developers to declare user-defined types, structs, and functions in a lightweight YAML specification. These user-defined constructs are automatically registered with the type checker and can be used in the definition of macros just like built-in types.

Finally, we integrated the type checker with the Language Server Protocol (LSP) [22] for the dbt VS Code extension [11]. This integration provides immediate feedback while developers edit models, surfacing type errors even before the full type checking is invoked. Features such as type hints and goto-definition are powered by our type checking engine. These capabilities bring the experience closer to that of statically typed programming languages, lowering the barrier to adoption and reducing the mental effort required to work with complex models. Together, these extensions make TypeJinja not just a back-end checker, but an interactive tool that supports developers throughout their workflow.

In summary, we make the following key contributions:

- ⋆ We designed and developed the first typed version of the Jinja language.
- ⋆ Our type checker is extensible and can be adapted as the set of types and functions evolve inside dbt.
- ⋆ We integrated TypeJinja into the dbt fusion engine, and used our analyses to empower the dbt VS Code extension.
- ⋆ We evaluated our type checker on a number of dbt Jinja models. We find that TypeJinja is efficient and it successfully discovered several issues in production models.



```
{% if a == 1 %}
    {{ a }}
{% endif %}
```

```
Ok((BlockStart,  @ 1:1-1:3))
Ok((Ident("if"),  @ 1:4-1:6))
Ok((Ident("a"),  @ 1:7-1:8))
Ok((Eq,  @ 1:9-1:11))
Ok((Int(1),  @ 1:12-1:13))
Ok((BlockEnd,  @ 1:14-1:16))
Ok((TemplateData("\n"),  @ 1:16-2:1))
Ok((VariableStart,  @ 2:1-2:3))
Ok((Ident("a"),  @ 2:3-2:4))
Ok((VariableEnd,  @ 2:4-2:6))
Ok((TemplateData("\n"),  @ 2:6-3:1))
Ok((BlockStart,  @ 3:1-3:3))
Ok((Ident("endif"),  @ 3:4-3:9))
Ok((BlockEnd,  @ 3:10-3:12))
```

```
Template {
    children: [
        EmitRaw {
            raw: "\n        ",
        } @ 1:1-2:9,
        IfCond {
            expr: BinOp {
                op: Eq,
                left: Var {
                    id: "a",
                } @ 2:15-2:16,
                right: Const {
                    value: 1,
                } @ 2:20-2:21,
            } @ 2:12-2:21,
            true_body: [
```

```
0: MacroStart(1, 1, 0, 3, 12, 39)
1: Lookup("a",  @ 1:7-1:8)
2: LoadConst(1)
3: Eq( @ 1:4-1:13)
4: JumpIfFalse(12,  @ 1:1-3:12)
5: EmitRaw("\n    ",  @ 1:16-2:5)
6: MacroStart(2, 5, 20, 2, 12, 27)
7: Lookup("a",  @ 2:5-2:12)
8: Emit( @ 2:5-2:12)
9: MacroStop(2, 12, 27)
10: EmitRaw("\n",  @ 2:12-3:1)
11: Jump(12,  @ 1:1-3:12)
12: MacroStop(3, 12, 39)
```

Figure 3: The workflow (top) and example (bottom) of the Jinja engine.

Our code is publicly available as part of the dbt fusion engine on GitHub: https://github.com/dbt-labs/dbt-fusion.

2 Background

In this Section, we provide a brief introduction to dbt (Section 2.1), Jinja (Section 2.2), and type checking for dynamic languages (Section 2.3).

2.1 dbt

Data Build Tool (dbt) is an analytics engineering framework that focuses on the *Transform* stage of the Extract, Load, Transform (ELT) workflow [6]. It allows dbt users to define a transformation pipeline of data in a warehouse and then orchestrates the pipeline in the correct order. By design, dbt does not move data out of the warehouse; instead, it renders transformation logic (SQL) from models, then executes the rendered SQL where the data already resides. This approach ensures scalability and takes advantage of the performance characteristics of modern data platforms.

At a high level, a transformation pipeline in dbt is organized into a *project*. A project consists of a configuration file, written in YAML, together with one or more models (which are placed in packages). The project defines the execution context, including schemas, connection details, and environment-specific options.

The central unit of work in dbt is the model. A *model* is typically a file that defines a transformation as a single query. These models are stored as code within a project directory. While at the surface definition of a model may look simple, the execution can become complex, depending on the size of the data, the warehouse characteristics, and the dependency structure among models. Models within a project can reference one another, forming a directed acyclic graph (DAG) that dbt uses to determine execution order. This structure enables clear data lineage, modular development, and consistent reuse across complex transformation pipelines. As projects evolve, models are frequently revised to reflect new business requirements and to improve efficiency.

Developers often use templating to keep models concise and flexible. For example, consider Figure 2, adapted from the dbt official documentation [9]. In the Jinja model in Figure 2a, in line 1 and line 2, we define a macro with its signature. We discuss signatures in more detail in later sections. On line 5, a loop will iterate over the parameter of the macro, generating one aggregation clause for every element in the list of `payment_methods`. This lets the developer maintain only a single model, even if the set of payment methods changes later.

Models are *rendered* by dbt into SQL queries (e.g., by running the `dbt compile` command). When the macro is called with parameter `payment_methods` = `['bank_transfer', 'credit_card', 'gift_card']`, the rendered SQL query is shown as Figure 2b. Line 3 corresponds to the aggregation over `bank_transfer`, and similar lines are produced for `credit_card` and `gift_card`. In this way, templating keeps the source model concise and reusable while still rendering SQL queries that can be executed directly against the data warehouse (e.g., by running the `dbt run` command).

2.2 Jinja

Within dbt, models are written using the Jinja templating language [36]. Jinja is a general-purpose templating engine for Python. It allows developers to embed variables, loops, conditionals, macros, and filters into otherwise static text files. In dbt, this capability is used to enrich models with dynamic behavior. When dbt executes a project (Figure 1), it first renders the Jinja models with the provided context and then executes the resulting SQL code directly against the data warehouse. This approach enables developers to adapt transformations to different use cases, reuse common SQL fragments, and keep their projects concise and consistent. Outside of dbt, Jinja is widely used in systems such as Flask [35] for generating HTML pages, Ansible [28] for configuring infrastructure, and Apache Superset [15] for writing parameterized SQL queries in dashboards.

The workflow of Jinja (which corresponds to box "Jinja rendering" in Figure 1) is illustrated in Figure 3. When a model is processed, the engine first performs ① *lexing*, turning the raw model string into a sequence of tokens that distinguish between literal text, expressions, and control structures. Next comes ② *parsing*, in which these tokens are transformed into an Abstract Syntax Tree (AST) representing the structure of the model. The parser output is then passed to a ③ *code generation* phase, which produces Jinja intermediate representation (IR) corresponding to the model logic. Finally, during ④ *rendering*, the instructions are interpreted with a specific context of variable bindings, producing the rendered SQL code.

The advantages of Jinja come from its ability to separate static structure from dynamic content. By isolating reusable fragments into macros and reusable blocks of template code, it reduces duplication and simplifies maintenance. Dynamic rendering allows models to adapt to diverse business requirements, deployment environments, or user inputs without requiring manual rewriting. In the context of dbt, these benefits translate into modular, parameterized, and easily maintainable models that can scale as projects grow.

2.3 Type checking in dynamic languages

Type checking [14, 25, 29, 30] is the process of verifying that values used in code are compatible with the types expected by macros, variables, or expressions. For example, a type checker ensures that a number is not combined with a string in an arithmetic operation and that a macro receives arguments of the correct types. This checks can take place before code execution through static analysis or at runtime when the code is executed. Some languages combine both approaches by allowing gradual addition of type information.

The main advantage of static type checking is the early detection of errors. Type mismatches can be caught before being executed, which prevents runtime failures and reduces debugging effort. Types also serve as lightweight documentation, making code easier to read and understand by clearly indicating expected inputs and outputs. As projects grow, type information provides a scaffold that helps developers maintain consistency across many components. Modern tools such as analyzers and editors can further use types to offer features like smarter autocompletion, error highlighting, and safer refactoring.

Languages such as Python and JavaScript are dynamically typed and therefore traditionally defer type checking until runtime. To improve reliability, their ecosystems have added optional static typing systems. In Python, developers can annotate code with type hints (using the `typing` package) and use tools such as mypy [13] to check the types before program execution. TypeScript is a typed superset of JavaScript that is compiled to plain JavaScript.

In dbt, Jinja models are rendered into SQL code that is executed against a data warehouse. In the specific case, type errors often only surface during execution of the rendered SQL. Applying static type checking in this context ensures that rendered SQL code is consistent with expectations and reduces the risk of subtle, hard-to-diagnose failures in production.

3 Example

To illustrate TypeJinja, we showcase an error that our type checker uncovered. The model, which is from dbt internal packages, as shown in Figure 4, defines an incremental materialization [8]. We leave out the details of incremental materialization, because it is not relevant for the work presented here, but the key takeaway is that the shown code is frequently used in practice. Errors in this model can therefore have direct impacts on dbt users.

In this example, the model calls `load_cached_relation(this)` (line 5) to retrieve an existing relation if it exists, or to return `None` if no such relation exists, as shown in the signature in line 1 in the above subfigure in Figure 4. Afterward, the model computes a

```
1 {# (relation) -> optional[relation] #}
2 {% macro load_cached_relation(relation) %}
```

```
3 {% materialization incremental, default -%}
4   -- relations
5   {%- set existing_relation = load_cached_relation(this)
        -%}
6   {%- set target_relation = this.incorporate(type='table'
        ) -%}
7   ...
8   -- configs
9   {%- set unique_key = config.get('unique_key') -%}
10  {%- set full_refresh_mode = (should_full_refresh() or
        existing_relation.is_view) -%}
11  -- trying to access attribute on a possibly None object
```

Figure 4: A bug in an internal dbt Jinja model: attribute access on a possibly None object.

configuration variable `full_refresh_mode` (line 10) by evaluating the expression `should_full_refresh` or `existing_relation.is_view`. The intention is to check whether a full refresh should be triggered or, alternatively, whether the existing relation is a view.

The subtle problem is that if `load_cached_relation` returns `None`, then the variable `existing_relation` does not refer to a valid object. As a result, the subsequent attribute access `existing_relation.is_view` attempts to read a field on a `None` value. When rendering, this causes a `NoneType` attribute error and the entire model fails.

Detecting this error through testing is non-trivial. A test would need to precisely simulate rare conditions where (1) the SQL dialect used in this dbt project is PostgreSQL, (2) the length of `this.identifier` is greater than 63 characters, under which the `load_cached_relation` macro returns `None`. Regression tests commonly do not cover these scenarios, which means the error can remain hidden until it unexpectedly surfaces in production.

Our type checker, TypeJinja, exposes this error statically before rendering. By reasoning about the possible return types of macro `load_cached_relation`, the checker infers that the variable `existing_relation` may be either of type `Relation` or `None`. When encountering the expression `existing_relation.is_view`, it identifies that one of the possible branches attempts an invalid attribute access on `None` and reports a type error. This error guides the developer to insert a defensive check, such as `if existing_relation is not None`, before accessing attributes.

This example demonstrates three key points. First, dbt models used in analytics pipelines can contain subtle errors that are easy to overlook during development. Second, such errors may have significant impacts, yet are difficult to detect through testing. Finally, a static type checker such as TypeJinja provides a lightweight and reliable way to catch these problems early, before they reach production. By integrating type analysis into the development workflow, we can prevent costly failures and ensure that dbt models are both correct and robust.

4 Approach

In this Section, we describe the design and implementation details of TypeJinja. Specifically, we cover our typed language (Section 4.1),

Notation. Γ is the typing environment (maps variables/attributes/functions to types);
Σ is the value stack; $T_1(\Sigma)$ is the top, $T_2(\Sigma)$ the second, and in general $T_i(\Sigma)$ the i-th from the top. Typing rules are stack-based: an instruction pops the required operand values from the top of Σ (e.g., $T_1(\Sigma), T_2(\Sigma), \ldots$) and pushes the produced value(s) back onto Σ.
A judgment $(\Gamma, \Sigma) \vdash \mathsf{Instr} : \tau$ states that, under Γ and Σ, executing Instr pushes a value of type τ onto the stack.
Environment updates are written $(\Gamma, \Sigma) \vdash \mathsf{Instr} \ \triangleright\ \Gamma'$; such rules update Γ but do not push values onto Σ.
Function types are $(\tau_1, \ldots, \tau_n) \to \sigma$; $\mathrm{typeof}(v)$ is the type of constant v.

$$\textsc{Add}\quad \frac{\begin{array}{c} T_1(\Sigma) : \tau, T_2(\Sigma) : \tau \in \Gamma \\ \tau \in \{Int, Float\} \end{array}}{(\Gamma, \Sigma) \vdash \mathsf{Add} : \tau}$$

$$\textsc{Sub}\quad \frac{\begin{array}{c} T_1(\Sigma) : \tau, T_2(\Sigma) : \tau \in \Gamma \\ \tau \in \{Int, Float\} \end{array}}{(\Gamma, \Sigma) \vdash \mathsf{Sub} : \tau}$$

$$\textsc{Eq}\quad \frac{T_1(\Sigma) : \tau \quad T_2(\Sigma) : \tau}{(\Gamma, \Sigma) \vdash \mathsf{Eq} : \mathsf{Bool}}$$

$$\textsc{Neg}\quad \frac{\begin{array}{c} T_1(\Sigma) : \tau \\ \tau \in \{Int, Float\} \end{array}}{(\Gamma, \Sigma) \vdash \mathsf{Neg} : \tau}$$

$$\textsc{CallFun}\quad \frac{\begin{array}{c} T_{1..n}(\Sigma) : \tau_1, \ldots, \tau_n \\ F : (\tau_1, \ldots, \tau_n) \to \sigma \in \Gamma \end{array}}{(\Gamma, \Sigma) \vdash \mathsf{CallFunction}(F, n) : \sigma}$$

$$\textsc{CallMet}\quad \frac{\begin{array}{c} T_1(\Sigma) : \tau_1 \quad T_{2..n}(\Sigma) : \tau_2, \ldots, \tau_n \\ \tau_1.F : (\tau_2, \ldots, \tau_n) \to \sigma \in \Gamma \end{array}}{(\Gamma, \Sigma) \vdash \mathsf{CallMethod}(F, n) : \sigma}$$

$$\textsc{Store}\quad \frac{T_1(\Sigma) : \tau \in \Gamma}{(\Gamma, \Sigma) \vdash \mathsf{StoreLocal}(x) \ \triangleright\ \Gamma[x \mapsto \tau]}$$

$$\textsc{Lookup}\quad \frac{x : \tau \in \Gamma}{(\Gamma, \Sigma) \vdash \mathsf{Lookup}(x) : \tau}$$

$$\textsc{GetAttr}\quad \frac{\begin{array}{c} T_1(\Sigma) : \tau \quad \tau.name : \sigma \in \Gamma \\ \tau : \mathsf{Struct} \end{array}}{(\Gamma, \Sigma) \vdash \mathsf{GetAttr}(name) : \sigma}$$

$$\textsc{SetAttr}\quad \frac{\begin{array}{c} T_1(\Sigma) : \tau_1 \quad T_2(\Sigma) : \tau_2 \in \Gamma \\ \tau_1 : \mathsf{Struct} \end{array}}{(\Gamma, \Sigma) \vdash \mathsf{SetAttr}(name) \ \triangleright\ \Gamma[\tau_1.name \mapsto \tau_2]}$$

$$\textsc{GetItem}\quad \frac{\begin{array}{c} T_1(\Sigma) : \tau_{idx} \quad T_2(\Sigma) : \tau_{base} \\ \tau_{base} : \mathsf{Tuple} \quad \tau_{base}[\tau_{idx}] : \sigma \in \Gamma \end{array}}{(\Gamma, \Sigma) \vdash \mathsf{GetItem} : \sigma}$$

$$\textsc{Const}\quad \frac{}{(\Gamma, \Sigma) \vdash \mathsf{LoadConst}(v) : \mathrm{typeof}(v)}$$

Figure 5: A subset of typing rules for Jinja IR instructions.

the type system (Section 4.2), the architecture (Section 4.3), and the extensibility (Section 4.4).

4.1 Typed language

We informally define our typing approach in this section before we formalize it in the subsequent sections.

Macro signatures. Each macro in TypeJinja is described by a signature of the form `(Type, ...)-> ReturnType` (see an example in Figure 2a, Line 1). Parameters may have explicit types, and the return type specifies the value produced when the macro is invoked.

Primitive types. Our type system supports a set of primitive types: Int, Float, Bool, String, Bytes, TimeStamp, and None. These types represent the types of primitive values encountered in models, such as numeric literals, booleans, string constants, byte sequences, timestamps, and `None` values.

Container types. Composite structures are expressed using container types. List[T] denotes a homogeneous sequence of elements of type T; Tuple[T_1,T_2,...] represents a fixed-size, heterogeneous sequence; Dict[K,V] represents mappings from keys of type K to values of type V; and Struct{...} represents a record with named fields, where each field has an associated type. These constructs allow representation of complex structural data.

Unions and optionals. Union types express that a value may belong to one of several alternatives, written A | B. Optionals are syntactic sugar for a union with None, so optional[T] is equivalent to T | None. These types are essential when model may yield different kinds of results along different control-flow paths.

Subtyping. The type system includes a small but useful hierarchy of subtypes. For example, `Struct` types are subtypes of more general ones if all fields in the general type are present in the specific type. Dictionaries with more specific key or value types can be used where more general ones are expected. This subtyping design enables reuse of macros and functions across a range of compatible data shapes without sacrificing type safety.

Overall, these constructs–primitives, containers, unions, and subtyping–provide a rich but practical foundation for checking dbt Jinja models. They allow TypeJinja to represent the data and control-flow patterns commonly seen in dbt projects while remaining lightweight enough for fast analysis.

4.2 Type system

Typing rules. We formally define typing rules for a stack-based Jinja IR abstract machine, that specify how each Jinja IR instruction manipulates types.

Figure 5 presents a subset of typing rules for core IR instructions. We only include a subset of rules due to the space limitations; the included subset illustrates key ideas behind our approach.

For example, the CallFun rule describes the type checking of a function call. F is the called function and n is the number of arguments to the function. The rule specifies that the argument types must match the expected parameter types, and the result type is determined by the function's signature.

As another example, consider the GetAttr rule, which describes the type checking for the IR instruction that gets the value of an attribute from the object on top of the stack. The rule finds the type of the object on top of the stack and then it takes the type of the attribute *name* from the type of the object.

In TypeJinja, multiple macros may share the same name, and resolution follows a deterministic search order: first within the current package, then in the root package of the project, and finally in the internal dbt package provided by dbt Labs. This design ensures that both users can override built-in macros and that common functionality can be shared across projects.

Checking. At the heart of our approach is instruction-level type checking, inspired by the Java Virtual Machine (JVM) bytecode verification algorithm introduced by Gosling and Yellin [17, 39]. In the JVM, type correctness is ensured through an abstract interpreter that simulates execution over types rather than concrete values.

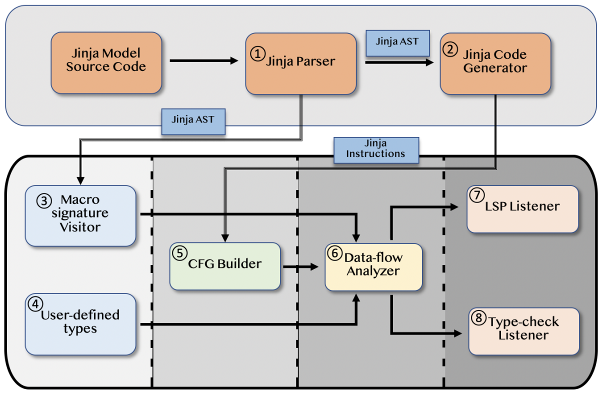


Figure 6: The architecture of TypeJinja, corresponding to the box "TypeJinja checking" in Figure 1.

We adopt a similar strategy for Jinja models. After generating IR instructions from models, we decompose the instructions into basic blocks and construct a control-flow graph (CFG). On this CFG, we run a forward data-flow analysis that computes the type environment at each program point. The typing rules defined in our system specify how each instruction consumes and produces values, and the checker ensures that every instruction is consistent with the expected types. Note that TypeJinja only verifies the internal type consistency of Jinja models; it does not validate the correctness of the rendered SQL.

When analyzing a model, a fixpoint computation is performed to propagate type environments across this CFG and merge information at join points. At join points, type environments are merged so that conflicting bindings are generalized to a safe supertype like `Union`. This design ensures that variables defined in one path but not in another are properly tracked, and that errors such as undefined variables are reported precisely. The use of the forward data-flow analysis allows us to cover all possible execution paths, ensuring that type consistency is maintained even in the presence of complex control flow.

To make the data-flow analysis precise, we formalize the propagation of type environments using the standard IN, OUT, GEN, and KILL [1] sets for each basic block b in the control-flow graph. The set IN$[b]$ represents the typing environment at the entry of b, and IN$[b_0]$ is initialized to the builtin typing environment which contains pre-defined variables and functions for the entry block b_0, while OUT$[b]$ is the environment at its exit. New type bindings introduced inside the block, such as variable definitions, are collected in GEN$[b]$, whereas invalidated bindings (e.g., variable redefinitions) are recorded in KILL$[b]$. The transfer function for each block follows the standard forward data-flow equations:

$$\text{IN}[b] = \bigcup_{p \in \text{Pred}(b)} \text{OUT}[p] \tag{1}$$

$$\text{OUT}[b] = \text{GEN}[b] \cup (\text{IN}[b] \setminus \text{KILL}[b]) \tag{2}$$

In adapting bytecode verification ideas to MiniJinja, we track variables, macro parameters, and expression types. Our rules incorporate the scoping model of Jinja, which includes local variables within macros, macros defined in other models, and global context variables. By handling these constructs, our type system enforces safety across the full range of MiniJinja features.

The typing rules also address common sources of subtle errors in templating. For instance, when an expression might yield either a scalar or a container depending on context, the checker enforces explicit handling of both cases. Similarly, when variables may be undefined on some execution paths, the analysis warns developers and encourages safe guards in the model logic.

4.3 Architecture

Figure 6 shows the architecture of TypeJinja. This figure corresponds to the box "TypeJinja checking" in Figure 1. As the figure shows, a Jinja model undergoes a multi-stage process before it produces the final rendered SQL. ① The model is first parsed into an AST that records its syntactic structure. ② This AST is then translated into IR instructions by the code generation component. Finally, the IR instructions are interpreted by the Jinja interpreter, producing the rendered SQL that downstream systems consume.

TypeJinja integrates directly into this workflow at the IR instruction level. This choice is deliberate: the IR serves as a compact, semantics-preserving representation of the model. Operating at this level ensures that TypeJinja captures the full expressive power of the templating language, including control flow and macro calls. This makes type checking both sound and portable across different dbt projects, which often vary widely in their data models and custom macros.

Before the data-flow analysis begins, the type-checker performs two preprocessing steps. ③ It first collects the type hints from macros defined in the models, since the signatures of macros are required for type checking. One example of the signature is shown in line 1 in Figure 2. ④ It reads from project-specific configuration files to collect user-defined types, structs, functions and register them into the type environment.

Next, after the model is parsed and IR instructions are generated, the type checker proceeds with its core analysis. ⑤ The instruction stream is segmented into basic blocks and organized into a CFG. ⑥ The data-flow analysis is then performed over this graph, propagating type information according to the typing rules defined in our type system.

By analyzing models immediately after instruction generation, TypeJinja can provide early feedback. Errors are reported before rendering, reducing the cost of rendering SQL and downstream debugging. More importantly, the integration is non-intrusive: developers do not need to modify their models or the dbt workflow. The results are exposed ⑦ to the dbt VS Code extension through the LSP listener or ⑧ to the command line interface (CLI) through the type-check listener, producing diagnostics that link directly back to source locations in the original model. This ensures that our contributions fit naturally into existing development practices.

4.4 Extensibility

A core requirement for TypeJinja is the extensibility. This is for two reasons. First, dbt is an evolving system with new functions, macros, and structs being added regularly. To remain useful, TypeJinja must be able to adapt to these changes seamlessly. Second, developers

```rust
pub struct AgateTable {
    repr: Arc<TableRepr>,
}
impl AgateTable {
    pub fn columns(&self) -> Columns {
        self.repr.columns()
    }
    pub fn column_names(&self) -> Vec<String> {
        self.repr.column_names().map(|s| s.to_owned()).
             collect()
    }
    pub fn rows(&self) -> Rows {
        self.repr.rows()
    }
}
```

(a) Rust implementation of AgateTable type.

```yaml
 1 ---
 2 object:
 3   id: agate_table
 4   attributes:
 5     - name: columns
 6       type: list[api.column]
 7     - name: column_names
 8       type: list[string]
 9     - name: __iter__
10       type: agate_table.__iter__
11     - name: rows
12       type: list[ANY]
13 ---
14 object:
15   id: agate_table.__iter__
16   call:
17     arguments: []
18     return-type: api.column
19 ---
```

(b) A user-defined type specification for agate_table in TypeJinja.

Figure 7: Implementation and type definition of AgateTable. (a) shows its Rust implementation; (b) shows the corresponding user-defined type used in TypeJinja.

who want to apply our type checker to other domains may need to define their own types and functions that are specific to their use cases. To remain useful in these settings, TypeJinja was designed with explicit support for extensibility.

In many projects, developers work with domain concepts such as `Table`, `Column`, or custom wrappers around strings and numbers. Figure 7a shows an example of a Rust struct that represents a table. These abstractions cannot be captured adequately by a small set of built-in types. To address this, TypeJinja allows developers to define user-specific types, structs and functions in a YAML configuration file.

Figure 7b shows an example YAML file that declares such types. Each `object` can be a struct or a method. In this case, two objects are defined (lines 2 and 14) in the YAML file. The `id` field (lines 3 and 15) is the identifier of the object; it serves as the fully qualified name by which other types can refer to it. Dots in an identifier (e.g., `agate_table.__iter__` on line 15) indicate a field or a method of an object. References such as `api.column` (line 18) denote a type from the field `column` in the `api` object.

Each definition specifies the type id, its fields or methods, and its relations to other types. These relations capture how objects connect to one another through attributes or function signatures. For example, a method may return another user-defined type (line 18), enabling rich interactions between custom types. When the type checker runs, these user-defined types are automatically registered and integrated into the analysis (Figure 6 step ④). As a result, models will be checked against these user-defined types rather than being restricted to built-in types. This design ensures that violations of project conventions are flagged early, and it empowers developers to encode domain knowledge directly into the type system.

5 Usage

We integrated TypeJinja into the existing dbt workflow in two ways: command-line interface and VS Code extension using the Language Server Protocol [22].

Command line interface (CLI). For batch analysis and continuous integration, TypeJinja provides a simple command-line interface through the existing dbt tooling.

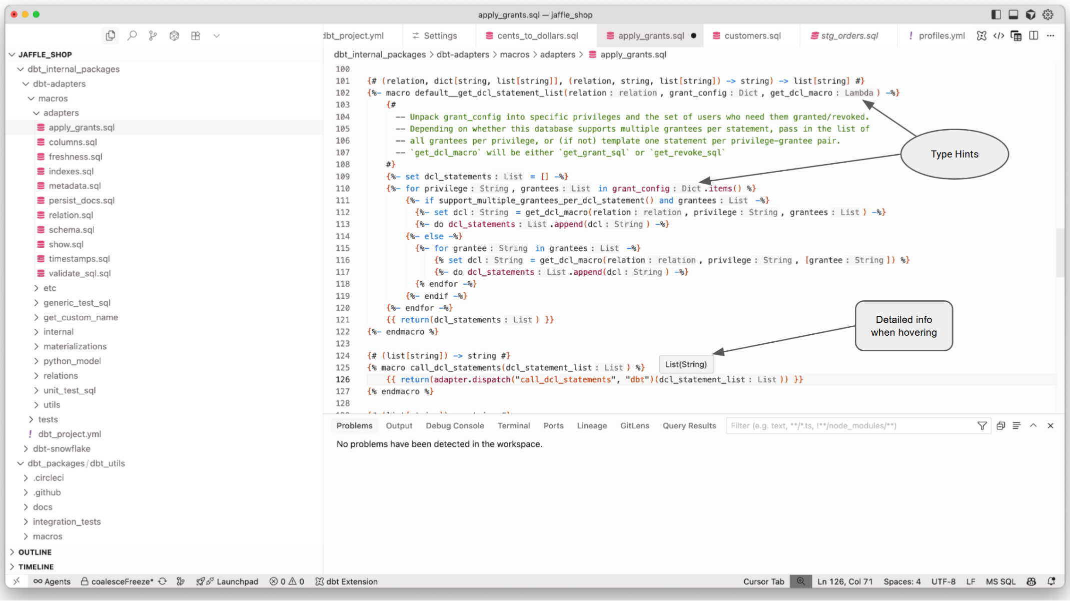
Developers can run `dbt jinja-check` command in any dbt project, and the tool automatically discovers all models, parses and analyzes these models, and reports any detected type errors. Each analysis result includes the file path, line number, and an explanation of the error, making it easy to trace problems back to their source. The CLI is lightweight enough to be integrated into CI/CD pipelines or pre-commit hooks, ensuring that new code is type checked before it is merged, while adding only negligible overhead to the build process.

Language server protocol (LSP) integration. Extensibility at the type-system level is complemented in the developer workflow. TypeJinja provides an integration through the LSP, enabling features available in modern programming environments directly inside editors such as VS Code. The LSP extension provides type hints, which annotate variables, macro arguments, and function results with their inferred types. These hints give developers immediate insight into the behavior of their models, reducing the cognitive overhead of tracking types manually. Figure 8 shows a screenshot of the VS Code LSP extension for TypeJinja.

By building on TypeJinja analysis, the LSP extension also supports goto-definition, allowing developers to navigate directly from a variable or macro usage to its definition. This feature is especially useful in large codebases where macros are frequently reused. Most importantly, the LSP integration provides analysis results in real time, surfacing type errors directly in the editor. Developers no longer need to run the full type check to see whether a model is valid: many issues are highlighted as soon as the code is written. Figure 8 demonstrates how these features appear in practice.

6 Evaluation

We evaluate our type-checking system, TypeJinja, in the context of dbt Labs's production environment. The goal is to assess its practical usefulness, scalability, and effectiveness in Jinja-based SQL pipelines.


apply_grants.sql — jaffle_shop
dbt_internal_packages > dbt-adapters > macros > adapters > apply_grants.sql
Type Hints
Detailed info when hovering
List(String)

```
{# (relation, dict[string, list[string]], (relation, string, list[string]) -> string) -> list[string] #}
{%- macro default__get_dcl_statement_list(relation : relation , grant_config : Dict , get_dcl_macro : Lambda ) -%}
    {#
      -- Unpack grant_config into specific privileges and the set of users who need them granted/revoked.
      -- Depending on whether this database supports multiple grantees per statement, pass in the list of
      -- all grantees per privilege, or (if not) template one statement per privilege-grantee pair.
      -- `get_dcl_macro` will be either `get_grant_sql` or `get_revoke_sql`
    #}
    {%- set dcl_statements : List  = [] -%}
    {%- for privilege : String , grantees : List  in grant_config : Dict .items() %}
        {%- if support_multiple_grantees_per_dcl_statement() and grantees : List  -%}
          {%- set dcl : String  = get_dcl_macro(relation : relation , privilege : String , grantees : List ) -%}
          {%- do dcl_statements : List .append(dcl : String ) -%}
        {%- else -%}
          {%- for grantee : String  in grantees : List  -%}
              {% set dcl : String  = get_dcl_macro(relation : relation , privilege : String , [grantee : String ]) %}
              {%- do dcl_statements : List .append(dcl : String ) -%}
          {% endfor -%}
        {%- endif -%}
    {%- endfor -%}
    {{ return(dcl_statements : List ) }}
{%- endmacro %}

{# (list[string]) -> string #}
{% macro call_dcl_statements(dcl_statement_list : List ) %}
    {{ return(adapter.dispatch("call_dcl_statements", "dbt")(dcl_statement_list : List )) }}
{% endmacro %}
```

```
No problems have been detected in the workspace.
```

Figure 8: The screenshot of the VS Code LSP extension for TypeJinja, including the type hints and detailed type information when hovering over variables.

Table 1: The performance measurements of TypeJinja on the evaluation projects.

Project	#Models	LOC	Time (ms)		
			T_{run}	T_{tc}	T_{pm}
jaffle-shop	178	6,869	306	54	0.303
IA	3,106	162,428	21,277	402	0.129
Overall	3,284	169,297	21,583	456	0.139

- T_{run}: end-to-end `dbt jinja-check` runtime;
- T_{tc}: time spent in type checking within that run;
- T_{pm}: average time spent in type checking per model.

Our evaluation is performed on a MacOS Sequoia 15.6 machine with an Apple M4 Pro CPU and 24GB memory.

6.1 Performance

To evaluate performance and scalability, we measured the runtime of TypeJinja when analyzing two dbt projects of different sizes, ranging from small open-source example to large production codebase that includes the dbt package containing a large number of reusable macros provided to dbt users. Table 1 shows the names of the projects (column 1)[1], the number of models (column 2), total number of lines of code across all the models (column 3), and time for running `dbt jinja-check` command, time for TypeJinja within that run, and average time for TypeJinja spent on analyzing each model (column 4).

In dbt-fusion, `dbt jinja-check` runs a sequence of phases including: *Debug*, *Deps*, *Parse*, *Format*, *Lint*, *Schedule*, *List*, *Freshness*, and *JinjaCheck*. The phases before *JinjaCheck* handle the tasks needed for the type checking phase, such as resolving dependencies, parsing the models, and formatting the code. TypeJinja is executed as part of this end-to-end workflow, so the total `dbt jinja-check` time (T_{run}) includes substantial costs beyond TypeJinja itself (T_{tc}).

TypeJinja successfully processed both projects without any crashes or timeouts. Even the large project, containing thousands of Jinja models, was type checked in less than one second. We also find that the overhead introduced by type checking is minimal compared to the overall runtime running `dbt jinja-check`. The per-model analysis time remained low across both cases, demonstrating that the approach scales well to production-scale workloads and can be integrated into existing CI/CD pipelines without much overhead.

6.2 Type errors detected in practice

The evaluated models were written by analytics engineers and cover a wide variety of use cases, from simple aggregations to complex incremental materializations [8].

The evaluation was conducted by running TypeJinja on the entire codebase and collecting all reported type errors. In total, TypeJinja reported 30 type errors, all of which were reviewed by dbt developers and confirmed as true positives. This shows that the tool is not only sound but also practically useful in uncovering real problems that had previously gone unnoticed.

For these projects, we *manually* added TypeJinja type annotations to the dbt macros used by the templates. We did not run automated inference to generate these annotations for the evaluation; however, in our experience the required macro annotations are lightweight for an experienced dbt developer, and we plan to explore automated annotation generation in future work.

We further analyzed the confirmed type errors and categorized them into three patterns:

[1]We intentionally do not show the actual name of the internal project.

```
{%- set existing_relation = load_cached_relation(this) -%}
...
{%- set full_refresh_mode = (should_full_refresh() or existing_relation.is_view) -%}
-- Type error: potential null dereference on existing_relation
```

(a) Null dereference on `existing_relation`: `existing_relation` could possibly be None while trying to access its properties.

```
{# (model, bool, relation, agate_table) -> string #}
{% macro reset_csv_table(model, full_refresh, old_relation, agate_table) -%}
...
{%- set old_relation = adapter.get_relation(database=database, schema=schema, identifier=identifier) -%}
...
{% set create_table_sql = reset_csv_table(model, full_refresh_mode, old_relation, agate_table) %}
-- Type error: passing optional relation to non-optional parameter
```

(b) Function argument mismatch: `old_relation` is an optional relation, however passed to a macro (see signature on the first line) which accepts a non-optional parameter.

```
{% set should_revoke = should_revoke(existing_relation, full_refresh_mode=True) %}
{% do apply_grants(target_relation, grant_config, should_revoke=should_revoke) %}
-- Type error: unknown local variable 'grant_config'
```

(c) Use of undefined variable `grant_config`: `grant_config` is not defined anywhere while being used.

Figure 9: Representative bug patterns detected by TypeJinja: (a) null dereference, (b) function argument mismatch, and (c) undefined variable. The error parts are highlighted in purple.

- **Null dereference** (17/30). Accessing attributes or invoking methods on variables that may be None, leading to runtime errors during SQL rendering.
- **Function argument mismatch** (12/30). Passing arguments of incorrect types to macros, that could result in runtime errors during SQL rendering or unexpected behaviors during execution of rendered SQL.
- **Undefined variables** (1/30). Using variables that are not defined in the current scope, that could result in runtime errors during SQL rendering.

Figure 9 shows representative examples of these patterns. The most frequent class of bug was null dereference (Figure 9a), often appearing in incremental models where a cached relation may not exist. Function argument mismatches (Figure 9b) typically arose in macros that were widely reused among models, and undefined variables (Figure 9c) appeared in a complex model with nested scopes. The distribution of bug types suggests that static type checking is particularly valuable in catching subtle runtime errors that are otherwise hard to detect with testing alone.

6.3 Discussion and insights

Our evaluation demonstrates that type checking in Jinja models is both performant and beneficial in practice. These results suggest best practices for template-based analytics pipelines. Developers could apply type checking early in their workflow to prevent latent bugs from propagating downstream. Macros and commonly used utility functions benefit the most from type annotations, as they serve as interfaces reused across many models.

7 Threats to Validity

Internal validity. A potential internal threat is that our evaluation may contain undetected implementation errors in the type checker itself, which could lead to false positives or negatives when reporting type errors. To mitigate this risk, we built extensive unit tests for the core typing rules and compared their behavior against known Jinja constructs. We also manually inspected a representative set of analysis results to confirm that the reported errors correspond to actual type mismatches. Another threat is that the realistic dbt codebases we analyzed might not cover every possible Jinja feature. We reduced this risk by selecting a large, production-quality project with diverse macros and control flow patterns, which exercises the majority of the language features our system supports, while some IR instructions that depend on runtime stack information remain unsupported. For example, the `unpacklists` instruction [12] pushes values onto the stack based on the length of the list found at runtime, which cannot be fully resolved statically. Moreover, at the moment, manual effort is required to mark the signatures of macros defined in Jinja models. Finally, TypeJinja might report false positives when models use macros that have no type signatures (which does not happen in our evaluation), and it might report no errors (false negatives) related to variables for which the type is not known at its declaration.

External validity. A potential external threat is that our findings may not generalize to all dbt projects or other templating environments beyond those we studied. However, our type system builds on the stable IR instruction representation produced by MiniJinja, which is designed to be semantics-preserving. Moreover, our support for user-defined types and macro signatures allows projects to tailor the checker to their own conventions, making the approach adaptable to a wide range of real deployments.

8 Related Work

We present related work in several areas: (a) templating languages, (b) type systems, (c) static analysis, and (d) hygienic macros.

Templating languages. Templating languages are used to mix text with dynamic content. They make it easier to build documents or web pages by letting developers insert variables and simple control structures such as loops and conditionals. Because of their convenience, templating languages such as Jinja are widely used in practice. However, this flexibility often comes at the cost of safety, since many errors in templates are only discovered when the program (which uses templates) is running.

Researchers have explored how to make safe templating languages. Heidenreich et al. [19] proposed rules for safe templating languages that prevent common runtime failures. In addition, several comparisons [7] show the wide variety of template engines available and highlight differences in their features and safety mechanisms. In contrast to prior work, which designed new templating languages with type safety, we focus on retrofitting static type checking into an existing, widely used templating language.

Type systems. A type system is a set of rules that assigns types to program elements, such as variables and functions, to help detect errors before the program runs. Over decades, many type systems have been proposed to make programs safer and easier to understand. In dynamic languages, type systems are especially challenging because programs may generate new behavior while the program is running (e.g., via metaprogramming). Ren et al. [29] addressed this challenge in Ruby with just-in-time static type checking, which defers the type checking after methods are created during runtime and before execution to catch errors early.

Gradual typing [16, 31–33] is another important line of work. It allows developers to start with dynamic typing and then gradually add static types. Phipps-Costin et al. [26] introduced a solver-based method to migrate dynamic typed programs towards static typing, while Moy et al. [24] proposed a system that combines gradual typing with static verification, moving some dynamic checks to static, which makes checks sound and efficient. Previous work also applies gradual typing to dynamic languages include Python [37] and TypeScript [27]. We differ from these works by focusing on type checking in a domain-specific templating language, which requires a type system designed for unique, domain-specific constructs.

Static analysis. Static analysis is the general term for techniques that study code without running it. The goal is to detect errors, enforce coding rules, and improve reliability before programs are executed. Unlike testing, static analysis can give guarantees about all possible executions of a program.

There has been extensive work on applying static analysis to dynamic languages. Grech et al. [18] developed preemptive type checking for such languages, allowing the earliest type error detection when an error is determined in some execution paths. Flanagan's hybrid type checking [14] combines static analysis and dynamic checking to enforce precise type specifications by statically verifying decidable cases and resorting to runtime checks for the undecidable ones.

More broadly, static analysis has been applied to many domains to discover potential problems early [2, 3, 40]. These efforts demonstrate the power of static analysis in practice, and they provide the foundation for adapting similar methods to template languages such as Jinja.

Hygienic macros. Hygienic macros are a macro expansion discipline that prevents *accidental capture* of identifiers: names introduced by a macro cannot unintentionally bind or shadow names from its use site, preserving lexical scope and referential transparency [4, 5, 21]. Early macro systems established algorithms that attach scope information to identifiers and systematically rename bindings during expansion to maintain hygiene [4, 21]. Pattern-based systems such as `syntax-rules` and more general `syntax-case` mechanisms have this behavior by construction, allowing macros to expand safely while respecting the scoping rules of the host language [5]. Subsequent research has sought to give hygiene a precise formal description and to extend it to richer settings. One line of work introduces explicit binding annotations to break the circularity between macro expansion and α-equivalence, proving that hygienic expansion preserves lexical scope as a semantic property rather than just an algorithmic artifact [20]. Comprehensive historical and technical accounts survey the evolution of hygienic macro technology and its adaptation to a range of languages beyond Scheme and Lisp [5]. TypeJinja macros provide limited scope isolation through local variables and explicit parameter passing, reducing some accidental name clashes but offering no syntactic hygiene or protection against capture in rendered SQL.

9 Conclusions

Jinja is a templating language widely used in many domains, including web development and data engineering. In particular, we have extensively used Jinja to enable dynamic SQL rendering in dbt, a popular data transformation tool. However the flexibility of Jinja templates can lead to runtime errors during rendering or when executing rendered SQL code, which can be costly to debug and fix. To address this issue, we present TypeJinja, a static type checker for Jinja templates. TypeJinja analyzes Jinja templates and their context to detect type errors before executing rendered SQL, providing developers with early feedback. TypeJinja is designed to be extensible, allowing easy type extensions as dbt evolves and enabling uses of the same framework for potential other uses of our version of MiniJinja. We have evaluated TypeJinja on our internal Jinja codebases and found 30 previously unknown type errors, demonstrating its effectiveness in improving code quality.

Acknowledgments

We thank Ivan Grigorik, Tong-Nong Lin, Aditya Thimmaiah, Zhiqiang Zang, Linghan Zhong, and the anonymous reviewers for their feedback on this work. We also thank many employees at dbt Labs for their comments and support. The first author was an intern at dbt Labs during Summer 2025. This work is partially supported by the US National Science Foundation under Grant Nos. CCF-2107291, CCF-2217696, CCF-2313027, and CCF-2403036. The views expressed are those of the authors and do not necessarily reflect those of sponsors.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc. https://doi.org/10.5555/1177220

[2] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE Software* 25, 5 (2008), 22–29. https://doi.org/10.1109/MS.2008.130

[3] B. Chess and G. McGraw. 2004. Static analysis for security. *IEEE Security & Privacy* 2, 6 (2004), 76–79. https://doi.org/10.1109/MSP.2004.111

[4] William Clinger and Jonathan Rees. 1991. Macros that work. In *Symposium on Principles of Programming Languages.* 155–162. https://doi.org/10.1145/99583.99607

[5] William D. Clinger and Mitchell Wand. 2020. Hygienic macro technology. *Proc. ACM Program. Lang.* 4, HOPL (2020). https://doi.org/10.1145/3386330

[6] Google Cloud. 2024. What is ELT? https://cloud.google.com/discover/what-is-elt.

[7] Wikipedia contributors. 2018. Comparison of Web Template Engines. https://en.wikipedia.org/wiki/Comparison_of_web_template_engines.

[8] dbt Labs. 2025. dbt Documentation: Incremental Materialization. https://docs.getdbt.com/docs/build/materializations#incremental.

[9] dbt Labs. 2025. dbt Documentation: Jinja Macros. https://docs.getdbt.com/docs/build/jinja-macros.

[10] dbt Labs. 2025. dbt Labs. https://www.getdbt.com/.

[11] dbt Labs. 2025. dbt VS Code Extension. https://docs.getdbt.com/docs/about-dbt-extension.

[12] dbt Labs. 2025. Unpacklists command implementation in dbt-Jinja. https://github.com/dbt-labs/dbt-fusion/blob/fcb812d2017f06f27e5edf96ca9ce65b75c79955/crates/dbt-jinja/minijinja/src/vm/mod.rs#L590.

[13] Mypy Developers. 2025. Mypy: Optional Static Typing for Python. https://mypy-lang.org/.

[14] Cormac Flanagan. 2006. Hybrid type checking. In *Symposium on Principles of Programming Languages.* 245–256. https://doi.org/10.1145/1111037.1111059

[15] Apache Software Foundation. 2025. Apache Superset. https://github.com/apache/superset.

[16] Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *Symposium on Principles of Programming Languages.* 429–442. https://doi.org/10.1145/2837614.2837670

[17] James Gosling. 1995. Java intermediate bytecodes: ACM SIGPLAN workshop on intermediate representations (IR'95). *SIGPLAN Not.* 30, 3 (1995). https://doi.org/10.1145/202530.202541

[18] Neville Grech, Julian Rathke, and Bernd Fischer. 2013. Preemptive type checking in dynamically typed languages. In *Theoretical Aspects of Computing.* 195–212. https://doi.org/10.1007/978-3-642-39718-9_12

[19] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, Christian Wende, and Marcel Böhme. 2009. Generating safe template languages. *SIGPLAN Not.* 45, 2 (2009), 99–108. https://doi.org/10.1145/1837852.1621624

[20] David Herman and Mitchell Wand. 2008. A theory of hygienic macros. In *Programming Languages and Systems.* 48–62. https://doi.org/10.1007/978-3-540-78739-6_4

[21] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming.* 151–161. https://doi.org/10.1145/319838.319859

[22] Microsoft. 2025. Language Server Protocol. https://microsoft.github.io/language-server-protocol/.

[23] Mitsuhiko. 2025. MiniJinja. https://github.com/mitsuhiko/minijinja.

[24] Cameron Moy, Phúc C. Nguyễn, Sam Tobin-Hochstadt, and David Van Horn. 2021. Corpse reviver: sound and efficient gradual typing via contract verification. *Proc. ACM Program. Lang.* 5, POPL (2021). https://doi.org/10.1145/3434334

[25] Tobias Nipkow and Christian Prehofer. 1993. Type checking type classes. In *Symposium on Principles of Programming Languages.* 409–418. https://doi.org/10.1145/158511.158698

[26] Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. 2021. Solver-based gradual type migration. *Proc. ACM Program. Lang.* 5, OOPSLA (2021). https://doi.org/10.1145/3485488

[27] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & efficient gradual typing for TypeScript. In *Symposium on Principles of Programming Languages.* 167–180. https://doi.org/10.1145/2676726.2676971

[28] Inc. Red Hat. 2025. Ansible: a radically simple IT automation system. https://github.com/ansible/ansible/.

[29] Brianna M. Ren and Jeffrey S. Foster. 2016. Just-in-time static type checking for dynamic languages. In *Conference on Programming Language Design and Implementation.* 462–476. https://doi.org/10.1145/2908080.2908127

[30] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type checking with open type functions. In *International Conference on Functional Programming.* 51—62. https://doi.org/10.1145/1411204.1411215

[31] Jeremy Siek and Walid Taha. 2007. Gradual typing for objects. In *European Conference on Object-Oriented Programming.* 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

[32] Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop.*

[33] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined criteria for gradual typing. In *Summit on Advances in Programming Languages.* 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

[34] Edgewall Software. 2024. Trac: an enhanced wiki and issue tracking system for software development projects. https://trac.edgewall.org/.

[35] Pallets Team. 2025. Flask Documentation. https://github.com/pallets/flask.

[36] Pallets Team. 2025. Jinja Documentation. https://jinja.palletsprojects.com/en/stable/.

[37] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and evaluation of gradual typing for Python. In *Symposium on Dynamic Languages.* 45–56. https://doi.org/10.1145/2661088.2661101

[38] Inc. VMware. 2023. Salt: Software to automate the management and configuration of any infrastructure or application at scale. https://www.saltproject.io/.

[39] Frank Yellin. 1995. Low level security in Java. In *International Conference on World Wide Web.* 369–379. https://doi.org/10.1145/3592626.3592656

[40] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J.P. Hudepohl, and M.A. Vouk. 2006. On the value of static analysis for fault detection in software. *Transactions on Software Engineering* 32, 4 (2006), 240–253. https://doi.org/10.1109/TSE.2006.38