Trustworthy-by-Construction Agentic APIs

Mark Marron

University of Kentucky
mark.marron@uky.edu

September 3, 2025

Overview

Agentic AI systems are intended to operate in, and interact with, the real world to accomplish tasks on our behalf. This includes tasks that are mundane, like searching for a recipe, but also tasks that involve destructive (or irreversible) actions, tasks that involve sensitive data, or tasks that entail financial or reputational costs to rectify if they are done poorly. In this world merely saying that a system is *likely* to behave correctly is not sufficient. Existing tool use frameworks [6, 25, 31] provide agents unchecked access to information and systems, which if misused, can have serious consequences. Guarantees about behavior are limited to a statistical statement that, under normal operating conditions, the system will, with high likelihood, behave as expected. This is insufficient for applications which handle sensitive data or perform important operations and, in the presence of malicious actors who seek to confuse or mislead an agent, this risk is unacceptable. This proposal aims to provide a framework for creating trustworthy-by-construction agentic APIs that allow AI agents to interact with the world in a safe and predictable manner even under unexpected (or adversarial) operating conditions.

The Bosque programming language [22, 23] and software stack are uniquely suited as a basis for this task. The Bosque language was specifically designed to support automated program analysis and validation as well as to serve as a target language for large-language model based code generation [22]. Using this stack as the basis this proposal aims to extend the model, and validation tooling, to create an API specification language ecosystems that is optimized for AI agent use, that allows developers to specify end-to-end safety/correctness properties on these APIs, and when given a specific use by an agent, to prove that the use meets the specified properties. This system will enable the construction and operation of highly dependable AI Agents. This framework will also create new opportunities for workflows with iterative self-analysis and correction as well as opportunities for improved training or synthesis via detailed feedback (reward) on the correctness of proposed actions.

Keywords: Automated Verification; APIs; Trustworthy Agents

Intellectual Merit

The proposed research addresses a foundational problem, ensuring predictable and safe behaviors, in the development of Agentic AI systems. The use of formal methods and programming language design is a novel contrast to the majority of work in the area which focuses on statistical guarantees and topics of training, prompting, or actor/critic designs. Beyond the merit of the overarching problem, this project will specifically advance the state-of-the-art in (1) understanding the interaction of programming language design and associated impacts AI Agent effectiveness in accomplishing

tasks and (2) opportunities for using formal methods in reward design for reinforcement learning and how this impacts ability to learn short and long horizon policies.

Broader Impacts

The research in this proposal has broad ranging societal implications. The ability to create trustworthy-by-construction APIs that allow AI agents to interact with the world in a safe and predictable manner is a foundational problem. Guaranteed trustworthy agents represent a critical advance over current systems which operate with statistical likelihoods of correctness and safety. The results will be reported in publications, talks, and collaborative open-source software. The project will involve research opportunities and training for undergraduate/graduate students. The PI has extensive interaction and contacts with the FinTech and Software industries. Artifacts produced in this project will be open-sourced and used as a basis for industry-wide collaboration and development.

1 Introduction

Agentic AI systems work to accomplish tasks on our behalf by interacting with the real world. Sometimes these tasks are mundane, like searching for a recipe or writing a little poem. However, other uses involve more complex interactions and involve potentially destructive (or irreversible) actions. In some cases these tasks may also involve managing sensitive data or sending/receiving messages with third-parties. In these cases the cost of an accidental failure or malicious actor seeking to confuse or mislead an agent is significant.

Existing tool use frameworks, *e.g.* the recently released *Model-Context-Protocol (MCP)* [25], provide unconstrained agent access to information and systems, which if misused, can have serious consequences. Guarantees about behavior provided are limited to a statistical statement that, under normal operating conditions, the system will, with high likelihood, behave as expected. This is insufficient for many applications which handle sensitive data or perform important operations and, in the presence of malicious actors who seek to confuse or mislead an agent, this risk is unacceptable. This proposal aims to provide a framework for creating reliable-by-construction agentic APIs that allow AI agents to interact with the world in a safe and predictable manner under unexpected operating conditions and/or adversarial situations.

This proposal takes the position that a foundational aspect of AI Agents is the interface they use to interact with the world. Specifically:

1. Agents use software APIs to interact with the world.
2. Agents must be able to reason, probabilistically *and* formally, about their actions and the potential consequences.
3. The design and specification of APIs must be optimized for Agentic use and analysis.

Based on previous work [21, 22] and industrial experience while working on Microsoft's original Copilot [14] project and with the Developer Tools Division [34], we hypothesize that, these three components are fundamentally linked. Fundamentally, an API that is easy for a human to use, and for formal verification tooling to reason about, is also an API that is easy for an *Large Language Model (LLM)* [32] AI Agent to use.

1.1 AI Agentic APIs

Fundamentally, accomplishing tasks in the world requires an agent to engage with external systems in some way. Most often this is done via calling a tool or service of some kind. For example, an AI agent might call a weather API to get the current temperature or a payment API to process a transaction. The manner in which these APIs (tools) are exposed to the agent is critical to the agent's ability to use them effectively. If the API lacks sufficient information, or is too complex, the agent may struggle to use it correctly.

We can also consider the resilience of an API to misuse or malicious engagement. An API that exposes too much information or that fails to verify the safety of an action can lead to unintended consequences. Exposing a database via raw SQL commands to an agent is a recipe for disaster, as the agent could easily execute a command that deletes all data or be tricked into exposing sensitive information. Just as with human developers, who have been known to accidentally enter the wrong number when executing a task, AI agents can make similar mistakes, and the same principles that apply to creating safe and easy to use APIs for humans also apply to AI agents!

Large Language Models (LLMs) have shown remarkable capabilities in generating code and reasoning about APIs [14, 32]. However, these models are foundationally text based, operating on code and available APIs, based on their representation as tokens. Thus, any properties of the APIs that are not explicit in the text available to the agent may be overlooked. Context engineering techniques [3, 10] can help, and automated context protocols [25] can provide a more structured way

to expose APIs, but these approaches are limited by the underlying API design. The less content in the specification the less the agent has to work with, and conversely, overly verbose specification systems lead to context confusion and agents that lose track of key information.

Thus, the first work item in this proposal is to develop an API specification system that is optimized for AI Agentic use. This work will draw on programming language design principles to develop a notion of lifting latent semantic API properties into explicit, and compact, syntactic features of the API for the agent to leverage. It will also include a focus on how to express useful sandboxing and guardrail properties that allow the agent to operate in a safe manner. The key objectives are to create a specification system that provides the highest likelihood of correct use by an agent, while also creating the foundation for later formal (and agentic) reasoning as well as dynamic validation of API use and training of agents.

1.2 API Validation and Agentic Reasoning

Given a task an agent will begin creating a plan, or script, to accomplish the task. This plan will include a sequence of API calls that the agent believes will accomplish the task and, particularly if these calls involve an irreversible action, the agent will want to ensure that the plan is safe and correct before executing it. A more advanced workflow would envision allowing an agent to reason about a partial plan, and if an API misuse is identified, to use this feedback online to correct the plan and continue execution. This is a key component of the Agentic programming model, where the agent is able to reason about its own actions.

To support this reasoning we need to provide a means for the agent to validate its API use. This can be done in several ways, the most direct of which is runtime validation of the conditions and resources used by the API. The Bosque language, and the *Agentic APIs* from the previous section provide, a unique environment for this as they fully sandbox the agent (preventing any surreptitious escapes) and provide a rich set of constraints that can be efficiently checked at runtime.

Recent developments using SMT solvers to reason about Bosque code have shown that it is possible to do fully automatic validation [24] of small to moderate sized programs! Thus, using the Bosque language, and the Agentic APIs, provides a unique opportunity to do complete static checking of a script or agentic plan to ensure that it satisfies all of the constraints and properties in the used APIs and any constraints the user might provide. This can be used to reject invalid or unsafe plans before any execution occurs, or to provide feedback to the agent about how to correct the plan. Preliminary experiments with this approach have shown substantial promise, increasing top-3 success rates for agentic plans by over 20%, even using naive feedback strategies without any additional fine tuning.

This generate-check workflow is a classic approach to program generation but fails to take advantage of the unique properties of AI Agents. The agentic ideal is for the agent to reason about the plans online as it is generating them, and to use the incremental feedback to correct and revise. This requires a more sophisticated approach to generation where the validation machinery is exposed to the agent as a tool that it can use, as one of many code generation actions, to create a script.

Thus, the second work item in this proposal is to develop a framework for runtime validation of API usage, extending the Bosque validator to handle all of the constructs present in the agentic APIs, and providing a means for the agent to use this validation machinery as a tool in its code generation process. This will include a focus on how to expose the validation machinery to the agent as one of many code generation tools. The key objectives are to create a framework that allows the agent to reason about its own actions, to validate its plans, and to use this feedback to correct and revise its plans in an online manner.

1.3 Learning and Using APIs

Solving long-horizon tasks requires an agent to be able to reason about its actions and the consequences of those actions. It also requires the agent to learn an underlying distribution, or policy, for what actions to take at each step. Thus, the final work item in this proposal is to develop a methodology for integrating the API validation and reasoning tools into the agent's training process. This will include developing a baseline for using direct reinforcement learning algorithms with tools [7] as well as exploring how to use the validation tools as a means for providing immediate feedback to the agent during training – allowing us to generate action rewards without a full trajectory and potentially drastically improving the process.

To support research in this area our first work item will be to develop a baseline set of tasks and performance to evaluate the impact of various tools on the agent's ability to complete tasks. One set of tasks we plan to use is the AppWorld [33] task set, which provides a rich set of tasks that require the agent to interact with a variety of APIs and services. Translating these APIs from their current form, text documentation and JSON schemas, into the Bosque API specs will provide experience with our specification language. We will also develop a set of tasks based on Shell activities and scripting. This second dataset represents a high-value domain for agents, has a different distribution of API behavior, and critically for us is a domain where incremental and interactive task completion is an appealing workflow. These datasets will provide an initial evaluation of the impacts of API design and access to tooling on the agentic performance.

With an appropriate task set in place we will then proceed to investigate the integration of validation tooling directly into reinforcement learning algorithms. This will include exploring the use reward shaping to drive the best use of validation tooling and immediate reward feedback from the validation tools to guide the agent's learning process toward quick conclusions instead of requiring extensive exploration. As the Bosque static validator can determine if a call will fail, or violate a policy even before execution, we can run it at intermediate steps in a rollout to determine if the current plan prefix is valid, and if not halt and propagate rewards immediately. We can also shape the reward to encourage the agents to use the validation tools effectively, such as by rewarding the agent for using the validation tool before making a call that would fail or violate a policy, and penalizing redundant user queries or tool uses. Interestingly, we can also identify cases where where an API call fails due to an unsatisfied precondition and what actions were missed. Thus, we plan to explore pseudo rewards even for actions that do not appear in a given trace with the goal of decreasing the number of needed learning steps and improving the agent's ability to learn from its mistakes.

Thus, the third work item in this proposal is to develop a framework for a fully integrated agentic API and tooling learning system. Our hypothesis is that the complimentary nature of the API specification, validation, and learning tools will allow us to create a system that is able to rapidly learn to accomplish tasks, adapt its behavior, and improve its performance over time.

2 Background on Bosque

The Bosque programming language and software stack is uniquely designed to provide a trustworthy-by-construction development model. Using this stack as the basis this proposal aims to extend the model, and validation tooling, to create an API specification language ecosystems that is optimized for AI agent use, that allows developers to specify end-to-end safety/correctness properties on these APIs, and when given a specific use by an agent, to prove that the use meets the specified properties. Thus, this system is well suited to support the construction and operation of highly dependable AI Agents using existing architectures.

2.1 The Bosque Programming Language

BOSQUE is not based on a single big feature, or even a number of small novel features, instead the value comes from a holistic process of simplification and feature selection with a single focus toward what will simplify reasoning about code. At the core of BOSQUE is a let-based functional language that is fully aligned with the goal of eliminating the complexities, mutablility, aliasing, inductive-invariants, and nondeterminism, that have historically limited the effectiveness of formal methods for program validation.

From this regularized core IR BOSQUE uses a variety of syntactic sugar to support commonly used and easily understood programming constructs like block-structured code, reassignment of variables, updates through (shallow) references, early returns, and object-oriented data types. These constructs are heavily used and well understood features in modern software engineering so supporting them allows developers to easily express their intents in a natural way. By careful construction, these features can all be compiled directly down into the core IR representation.

A simple BOSQUE program, Figure 1, provides a flavor of the language. The code implements a simple *sign* function. This code is very similar to the implementation one would expect in Java or TypeScript – in fact just eliminating the explicit `'i'` specifier on the literals would make it valid TypeScript.

```
1 function sign(x: Int): Int {
2   var y = 1i;
3   if (x < 0i) {
4     y = -1i;
5   }
6   return y;
7 }
```

Figure 1: Sign function in BOSQUE.

This function highlights the use of multiple updates to the same variable and block structured conditional flows. BOSQUE distinguishes between variables, let, that are fixed and those, var, that can be updated. This ability to set/update a variable as a body executes simplifies a variety of common coding patterns. This code can be easily converted to a SSA form [11], and the loop freedom of the language, ensures that any assignment then also a single dynamic assignment. Thus, reasoning about the code is entirely equational and the resulting SMTLib [8] encoding, Figure 2, of this code is almost a 1-1 translation of the BOSQUE version.

```
1 (define-fun sign ((x Int)) Int
2   (let ((y 1))
3     (ite (< x 0)
4       (let ((y -1)) y)
5       y
6     )
7   )
8 )
```

Figure 2: Sign function in SMTLib.

Since BOSQUE is loop free it needs to provide an alternative form of iterative processing. Empirically [1], and anecdotally, the vast majority of iterative processing is over collections and 90%+ of this processing can be done with higher-order functions *e.g.* Java Streams or C# LINQ. These higher-order functions are a powerful programming mechanism that can be used to great effect to simplify code as in Figure 3.

```
let l = List<Int>{1i, 2i, 3i};

let x = l.allOf(pred(x) => x >= 0i) %% takes a lambda pred -> true
let y = l.map<Int>(fn(x) => x + 1i) %% takes a lambda fn -> List<Int>{2i, 3i, 4i}
```

Figure 3: List and operations in Bosque.

As Z3 [19] and other SMT solvers have become more powerful, the ability to reason about higher-order functions has also improved[1]. Thus, we can transform this code into the SMTLib representation, shown in Figure 4, which again is a nearly 1-1 mapping of the Bosque code.

```
(let ((l (seq.++ (seq.unit 1) (seq.unit 2) (seq.unit 3))))
  (let ((x (seq.fold_left (lambda ((acc Bool) (vv Int)) (and acc (p vv))) true l)))
  (let ((y (seq.map (lambda ((vv Int)) (f vv)) l)))
  ...
  ))
)
```

Figure 4: List and operations in SMTLib.

2.2 Bosque Validation

The Bosque language includes builtin support for various validation workflows, including a specialized declaration of a parametric property test. A simple validation test that checks that the `sign` function returns a value in the range $[-1, 1]$ for any input (`x`) is shown in Figure 5. For a simple property like this the Bosque compiler can show that the property holds for all inputs and takes 7ms to complete.

```
...
chktest signRange(x: Int): Bool {
    let sgn = sign(x);
    assert /\(-1i <= sgn, sgn <= 1i);
}
```

Figure 5: Example validation harness in Bosque.

The Bosque validator is also able to go over a small application and check, for each possible runtime or user defined error, that either the error is impossible or that it can be triggered – and also generate a witness input. Results for three small sample applications are shown in Table 1, the first is a sample application from Morgan Stanley, the second is a raytracer demo from the Microsoft MSDN blog, and the third is a port of an in-memory database (DB) from the SpecJVM98 suite.

The results of this all-error checking are shown in Table 1. The first column is the name of the application, the second is the code that was loaded for analysis (including user and runtime code), the third is the number of error conditions from the user code that were checked. The last three columns are the total time for all checks, the maximum time taken for any single check, and the number of failures found.

The key aspect of these results is 1) that the analysis is fully automatic and does not require any additional lemmas, annotations, or proof support 2) that the analysis is fast enough and consistent enough to be run on the scale of code generated by an AI agent. These features are critical to the

[1]Sequences with lambdas are fully decidable with bounded input sizes and lambdas in direct position.

Application	Total Lines	Checked Errors	Total Time	Check Max	Failures
market	0.9 Kloc	6	50ms	12ms	0
raytracer	1.5 Kloc	15	422ms	54ms	1
db	1.8 Kloc	24	1359ms	135ms	1

Table 1: Results of the all-error validation on sample applications.

practical use of formal validation in the context of this work and a unique, new capability, that the use of BOSQUE provides in this problem space.

3 Research Plan

Our research plan is focused on a three-pronged approach to creating trustworthy AI Agentic systems. Each of component plays a complimentary role in specifying, validating, and learning how to use APIs in a way that is both safe and effective: *API Design Optimized for Agents* via a novel API specification language and system event logic (Theme 1, §3.1), *Validation of Actions on the API* via extensions to the BOSQUE verification stack (Theme 2, §3.2), and *Leveraging Feedback to Train and Improve Agent Actions* via integration of API constraints and validation tool at training time and as oracles for agents to query interactively during planning (Theme 3, §3.3).

3.1 Theme 1: Agentic Optimized Specifications and Checkflow Logic

In this theme our goal is to explore the design of an API specification language that is optimized for AI Agentic usage and that allows the direct specification of behavioral guarantees. Results from previous investigations into AI aware programming language design [22] and Data Specifications [23] demonstrated the effectiveness of this general approach and provide a foundation for this investigation. This proposal envisions extending these ideas from pure and isolated type (or function) specifications to systems where actions are effectful and specifications range over sequences of actions and events.

The first step in creating a truly Agentic optimized programming API framework is to develop a suitable action language and specification system. Consider the task of paying a bill. The core API is as simple as:

```
1  type USD = Decimal;
2
3  entity Account { ... }
4  entity Confirmation { ... }
5
6  api transfer(amt: USD, from: Account, payee: Account): Confirmation;
```

This API provides a syntactically explicit and useable description for an AI Agent. BOSQUE is designed to move information from implicit (API documentation pages) into explicit syntactic features of the code. In our example the type system allowing the type alias of USD to be used as a unit of currency instead of a simple number with a comment that it is a dollar amount. Preliminary experiments show that, even without specific training, an model like Gemini-2 or GPT-4 has a roughly 20% higher success rate of generating code using this API than the equivalent in TypeScript.

Even with this significantly improved success rate the API is still ripe for accidental misuse or targeting by malicious actors seeking to confuse our agent. Today we could improve this API in BOSQUE by adding a simple pre-condition to the API:

```
api transfer(amt: USD, from: Account, payee: Account): Confirmation
    requires 0.0<USD> < amt;
    requires amt <= 100.0<USD>;
;
```

These simple pre-condition ensure the useful constraint that we should never pay a negative amount of money (also a payment of zero does not make sense) and that the API cannot be used to transfer more than $100.0 USD. However, this is not entirely satisfactory as an agent could still erroneously transfer money, even if the amount is small. Further, the dollar amount is hard-coded into the API and does not allow for situational flexibility when the agent is operating in a different contexts/environments.

3.1.1 An Explicit Agent Environment and Resource Model.

Fundamentally, AI Agents are always operating in some context or environment. This context may be information about the current state of the world, such as the current location, a most recent text message list, or the name of the person the agent is acting on behalf of, or this information may describe resources/constraints available to the agent, such as a database authorization token, an allow-list of accessible files, or a limit on spending. In our example, we would like to be able to include this context in the API specification so that the agent may use this information to make decisions about how to use the API and we can verify that the agent is behaving correctly with respect to this context.

Conveniently, programming languages have two well known concepts that support these needs – *Scoped Dynamic Environments* and *Uniform Resource Identifiers* (URIs). Scoped Dynamic Environments allow us to describe a set of ambient variables that are available to the agent during the execution of some code and to define what context needs to be setup to successfully use and API. The ambient nature of these variables provides a natural way to descibe the context in which an agent is operating while dynamic scoping allows us to extend and modify the context as the agent operates. URIs are a familiar and flexible (syntactic) way to describe permissions at a logical level – notably the underlying data representation is not exposed via the URI naming scheme which is simply a logical syntactic identifier. Thus, developers can use these to organically create a model of arbitrary resources and access controls (using Globs) that compactly describe the resources that an agent can access (and that are accessed by any given API call).

```
api transfer(amt: USD, from: Account, payee: Account): Confirmation
    env={
        PAYMENT_AUTHORIZATION: OAUTH_TOKEN,
        PAYMENT_LIMIT: USD
    }
    permissions={
        \account:${from.routing}/${from.account}\
    }

    requires 0.0<USD> < amt;
    requires amt <= env.PAYMENT_LIMIT;
;
```

Figure 6: An example of an API specification that uses scoped dynamic environments and URIs to describe the context in which the API can be used.

Figure 6 shows an example of how we can use these concepts to create a more flexible API specification. The API now describes the context in which it can be used, including the required OAuth token for payment authorization and the maximum payment limit. The API also describes the resources the API accesses, in this case the account that the payment is being made from.

In addition to providing a more flexible API specification, this approach also allows us to more precisely specify the behavior of the API. The pre-conditions now refer to the dynamic environment variable `env.PAYMENT_LIMIT`, allowing the specification to be parameterized by the current context. In addition the permission URIs can be specified, not just as constants, but parametrically based on the values of the environment or, in this case, the values of the API parameters.

An interesting feature to note in the specification from Figure 6 is that the API `permissions` set only includes the `from` account. Although this may seem counter-intuitive, it is a natural consequence of the logical model of URIs as permissions and not physical resource requirements. Although the API is transferring money from one account to another, and physically the `payee` account will likely be modified eventually, the API does not need to have permissions to access to the `payee` account in order to execute the call to the transfer API. This highlights the distinction between logical permissions and physical resource access in the API design. This model allows the creation of permissions that are flexible and can be developed organically for domains as needed as opposed to requiring the creation of a fixed set of resource types and capabilities [2] which developers have historically struggled to manage.

This API specification has several features that are natural improvements on existing state-of-the-art approaches in the Agentic space. When looking at Model-Context-Protocol (MCP) [25], and the underlying concepts from REST [13], which envision dynamic discoverability and use of APIs, we see that the proposed Bosque specification language and extensions provide several improvements. First, as opposed to the verbose and limited type system provided by JSON Schema/OpenAPI, Bosque provides a rich type system that allows for more precise and expressive API and compact specifications. This increases the likelihood of successful correct use by an agent and reduces the number of context tokens used – which reduces the risk of dilution in the prompt. Additionally, the structured nature of the components of the specification provides a strong structure for the underlying LLM to key on. In the payment example, the specifications make it explicit that the API enforces a payment limit, *e.g.* the `requires` clause and the code for the check, are strong signals to the agent that it needs to consider the relation between the payment amount and the limit before calling the API.

The explicit nature and structure of the specifications, as opposed to free form text structure in MCP, also simplifies the tasks of pre-selecting tools (APIs) via Resource Augmented Generation (RAG) [20] and improves the ability of an agent to correctly select which tools (APIs) to use for a given task. This is particularly important in the context of adversarial attacks, where an agent may be tricked into using an API that is not appropriate for the task at hand. In RAG a pre-processing document retrieval step is used to search for the most relevant items, in our case APIs, to include as possible resource for use in the agent's task. The structured nature of the Bosque API specification allows for more precise and effective retrieval of relevant APIs. Once in the agent's context, the structured nature of the API specification allows the agent to more easily reason about the APIs and which are the best options. In the payment example, the specification is explicit about the permissions, allowing the agent to quickly ignore APIs, that it does not have the needed permissions for.

Task 1.1: Static API Specifications. *The objective of Task 1.1 is to develop and experiment with extensions to the* Bosque *language for specifying Agentic APIs and generate a baseline success rate over a set of benchmark tasks.* The results of this work will provide a baseline for future investigation as well as generate a solid foundational Agentic API specification language that is capable of effectively encoding static properties of APIs that are useful for AI Agents.

3.1.2 Behavioral Checkflow Logic.

The objective of Task 1.1 is to develop a specification language that allows for the direct specification of static behavioral guarantees. However, agentic tasks are dynamic multi-step processes that involve sequences of actions and events. The APIs that are used often have constraints on the order of their use or other temporal properties that must be satisfied. For example, a payment API may require that a user confirmation has been made before a payment is sent, or before sending a purchase confirmation, that the reservation has been successfully made in the system. If we look at the example payment API we can extend it with a user confirmation requirement as shown in Figure 7.

```
1  api transfer(amt: USD, from: Account, payee: Account): Confirmation
2      env={
3          PAYMENT_AUTHORIZATION: OAUTH_TOKEN,
4          PAYMENT_LIMIT: USD
5      }
6      permissions={
7          \account:${from.routing}/${from.account}\
8      }
9
10     requires 0.0<USD> < amt;
11     requires amt <= env.PAYMENT_LIMIT ||
12              $events.contains(ExplicitUserApprove{|payee=payee, amt=amt|});
13 ;
```

Figure 7: An example of an API specification wih a behavioral checkflow.

In this conceptual example we have added the requirement that the payment amount must be less than the limit, or that an explicit user approval event has occurred. The fundamental concept here, of temporal safety properties, and has been explored in many contexts [9,18,26,30]. However, these systems have several limitations in this application. The first is that temporal logics are often difficult for developers to author and are limited in the range of properties that can be expressed. The second is that they introduce a second language and logic into the system, which increases the complexity of understanding and authoring a specification.

Instead for this proposal we envision a more direct and intuitive way to express these properties using the Bosque language. Based on extensive developer interviews and conversations, while working on developer tools at Microsoft and Morgan Stanley, we have found that developers often think of these properties in terms of a (linearized) series of events that occur during the execution of a system. Then conditions over the sequence of events, such as "the user must approve the payment before it is sent" or "if a read etag matches the previous write etag then the contents of the read are equal to the contents written", can be expressed in a more natural way as code checking a boolean condition on the event list. This has the benefit of being directly expressible as code in a manner a developer (or LLM based agent) is already familiar with and allows for the expression of arbitrary computable conditions over the sequence of events.

This form of logical predicates over sequences of events can also be used to express the effects of API calls as well. Consider an application for renting a sailboat. A sailboat rental API might have a sequence of events that includes checking the weather forecast and confirming inventory availability. An example of such an API from a Morgan Stanley tech-demo is shown in Figure 8.

In the example (Figure 8) we have an API that allows an agent to rent a sailboat for a given day. The API has two permissions, one for the weather service and one for the internal sailboat availability service (which is otherwise opaque to us). The API is defined as ensuring that if the response is a success, then two events must occur during the execution of the API call:

```
datatype Response = Success | Failure;

api rentSailboat(quantity: Nat, day: LocalDateTime): Response
    permissions={
        \web_http:api.weather.gov/\,
        \ms_service:sailboat_availability\
    },

    ensures Response === Success ==>
            $events.contains(SafeWeather{|day=day, response=Ok|});
    ensures Response === Success ==>
            $events.contains(ReserveInventory{|
                day=day, quantity=quantity, response=Approved
            |});
;
```

Figure 8: An example of an API specification wih a behavioral checkflow.

1. There must have been an API call to check that the weather is safe for sailing that day.
2. There must have been a successful API call to ensure availability and reserve the sailboat inventory for the given quantity.

By directly integrating these specifications in the language, and automatically emitting the event generation as part of the runtime, the system can track and verify these properties. These properties are also made syntactically visible to the AI Agents so that they can identify key requirements and effects to reason about. This should allow them to effectively identify which APIs are appropriate for accomplishing their objectives and to identify appropriate sequences of API calls to accomplish them (as well as what potential failures and recovery strategies are needed).

Task 1.2: Checkflow Logic and API Specifications. *The objective of Task 1.2 is to develop and experiment with a novel temporal logical extension to the* Bosque *language for specifying dynamic API properties.* The results of this work will provide a powerful new tools for specifying dynamic properties of APIs and will allow our AI agents to reason about the sequences of actions needed to accomplish their tasks.

3.2 Theme 2: Validation of Agentic APIs and Tools

In this theme our goal is to explore three mechanisms for ensuring that AI agents always use APIs correctly! The first step is to implement a comprehensive sandboxing and runtime-verification system in the Bosque runtime. This baseline safety guarantee will immediately allow us to deploy agents into environments where correctness and security are critical. The second step is to integrate a formal validation system that allows us to prove that an agent's use of APIs (and thus behavior) is correct *w.r.t.* to the API specification. This will allow us to provide strong guarantees about the agent's behavior in a wide range of scenarios even prior to execution and also has the potential to substantially improve the agents success rate in task completion. The third step is to expose the validation tools directly to the agent during their planning process as an online feedback loop. This will allow us to create agents that are able to reason about their own actions.

3.2.1 Runtime API Checking.

Runtime validation is a well studied problem in programming languages and systems [16, 18]. The specification language for APIs in Bosque presents several novel challenges as the API specifications may be arbitrary code, the specifications include sandbox constraints, and there are unique challenges in domains with agentic systems (*e.g.*, an agent using a shell [15]).

Sandboxing resources is a critical topic for agents as prompt injection attacks [25, 27] can be used to exfiltrate information from the agent. Recent work has looked at reviving classic resource

sandboxing techniques for Java [2] or filesystem permissions [15]. However, as described in §3.1.1, these access models have historically had limited adoption with developers in practice. Thus, this proposal will explore option for the runtime validation of the parametric URI path based resource sandbox specifications. As this is a novel approach our first step will be to evaluate the performance implications of API calls with a naive check of the paths at each call. Based on the outlined design, checking if a given API call is compatible with the resource constraints requires checking path constraints of both the current API at runtime, our hypothesis is that the URI path constraints should be linear time checkable which would ensure that the cost of the check is minimal.

For the `requires` and `ensures` clauses, as these are simply developer written BOSQUE code, we believe that users will be able to anticipate the costs of running these checks dynamically or be able to profile and adjust them using standard engineering practices. One of the most common patterns is to selectively enable these checks in production environments, *i.e.* test or debug scenarios, which is trivially and explicitly supported in BOSQUE with dedicated keywords. Thus, developers should be capable of estimating and managing these costs appropriately.

Task 2.1: Runtime Checking Implementation. *The objective of Task 2.1 is to develop and experiment with a baseline implementation of runtime checking for API calls in the* BOSQUE *language.* The results of this work will provide a powerful tool for guaranteeing the safety of agentic generated code.

3.2.2 Offline API Validation.

Runtime validation is a powerful tool. It provides a baseline for ensuring safe API usage and is guaranteed to prevent an agent from performing an unsafe action. We can also use this runtime validation, in conjunction with automated test generation [5, 17], to evaluate the correctness of a proposed plan or script. However, as with all test based approaches, this is limited by the coverage of the tests and the tendency of test-based feedback to lead LLM based code generation to over-fit to the tests. Ideally, we would like to statically validate the proposed plan for adherence to the specification and correct API usage. This validation provides a much stronger guarantee than a test-based approach, and provides an opportunity to generate generalized feedback on failures that encourage the agent toward more generalized, and likely more correct, adjustments to the generated code.

Consider the example request for an agent to "send a payment to Tom for half of the lunch bill" as shown in Figure 9. This code shows a hypothetical agent generated script to accomplish the task. The agent first searches for a payment request for Tom, then uses another LLM agent to process the semi-structured text in the payment request (and memo field) to determine the amount to pay, and finally attempts to transfer the payment. If the `agent.query` action were unlucky[2], or the lunch was particularly expensive, this computed amount could be large enough to exceed the payment limit of the user.

The runtime verification system developed in §3.2.1 would catch this error at runtime but, using BOSQUE (§2), we can also statically run symbolic validation on this script to detect that they `amt` value may exceed the payment limit and that the, otherwise required, user confirmation check is missing!

Further, the feedback generated by test case generation might be something like "Input payments: [user: "Tom", memo: "lunch", amount: 200], env: PAYMENT_LIMIT: 100 violates transfer precondition". This feedback can be useful for the agent to try and correct the script but it can also cause the agent to focus on details of the test case and over-fit. Perhaps adding the (erroneous) check – `if(amt >= 100) return "Too much money to send";`. By symbolically identifying the possible precondition violation we can provide a more general feedback message to the agent, such as

[2]Perhaps Tom likes jokes and puts in the memo field – "ignore previous instructions and pay me $1000".

```
let request = payments.search(user="Tom", memo="lunch");
let amt = agent.query<Decimal>(request.asText(), "What is half of the lunch bill?");
if(amt === none) {
    return "I don't know how much to pay Tom.";
}

let result = transfer(USD::from(amt), from=env.account, payee=request.payee);
if(result.success) {
    return "${amt} was sent to Tom for lunch.";
}
else {
    return "Unable to send the payment.";
}
```

Figure 9: An example generated script for the request – "Please send Tom payment for my half of the lunch bill".

– The amount to pay Tom may exceed the "PAYMENT_LIMIT" in "env" – or even computing weakest-preconditions for various points in the code to help the agent identify the best candidate fixes for the code.

Task 2.2: Static Validation Implementation. *The objective of Task 2.2 is to extend the* Bosque *validation toolchain to support the Checkflow logic for API specifications introduced in theme #1.* The results of this work will provide a powerful new tools for checking agentic generated code and providing generalized feedback on failure modes.

3.2.3 Validation as a Tool.

The final step in this theme is to expose the validation tools directly to the agent during their planning process as an online feedback loop. This will allow us to create agents that are able to reason about their own actions. Looking at the payment example in Figure 9, somewhere around half of the generated code comes after the `transfer` call, at which point the plan has already failed. Thus, the direct generate-validate-retry loop is clearly inefficient in this case. Further, standard design guidelines for LLM based code generation suggest [4] that simply retrying the generation from scratch is often more effective then attempting to ask for a correction to the existing code – which often results in the agent performing other error introducing operations in addition to fixing the issue that was flagged.

Instead, this proposal intends to explore how to integrate validation tools directly into the agents planning and code generation process. Tool use is an active topic of investigation, with the most common tools being abilities to run tests, search codebases, make notes, and edit code [14, 25, 32]. These agents proceed by generating code interleaved with tool uses.

In our example we envision the agent being able to run the validation tool before an API call is added to the partial plan to see if the call is valid and if not, what needs to be done before the call can be made.

The code in Figure 10 shows a hypothetical agent & tool chain-of-thought, tool use, and response. At the point where the agent is preparing to call an API, *e.g.* the `transfer` API, it can call the validation tool to check that the requirements for the API call are met. The validation tool can then respond with a list of missing checks or requirements that the agent needs to address before the API call can be made. Using this feedback the agent can either emit the action code, if everything is satisfied, or generate additional code to address the missing requirements.

This capability should enable the agent to achieve a higher success rate in task completion and be more robust to errors, as it can reason about its own actions and correct them before they are executed. Further, this approach has the potential to enable a new class of reactive, notebook style,

```
let request = payments.search(user="Tom", memo="lunch");
let amt = agent.query<Decimal>(request.asText(), "What is half of the lunch bill?");
if(amt === none) {
    return "I don't know how much to pay Tom.";
}

%% Agent -- I want to call the transfer API with amount set to amt
%% Agent -- calling validation tool to check that API conditions are met
%% Response -- Missing check for PAYMENT_LIMIT or explicit user confirmation
```

Figure 10: An example of online agentic generation with validation as a tool.

agents that interleave code generation, validation, execution, and user interaction, to accomplish complex tasks in a more efficient and effective manner.

Task 2.3: Exposing Validation Tools to Agents. *The objective of Task 2.3 is to develop and experiment with a methods for exposing validation as tools that an agent can access online as it generates code.* The results of this capability should lead to substantial increases in the success rate of agentic code generation and a more robust, and possible interactive, agentic systems.

3.3 Theme 3: Learning to Reason About and Use APIs

Solving long-horizon tasks requires an agent to be able to reason about its actions and the consequences of those actions. It also requires the agent to learn an underlying distribution, or policy, for what actions to take at each step. Running an analysis before making an API call that has destructive or irreversible effects is crucial for ensuring safe and correct behavior. Querying a user to confirm an action before taking it or asking for additional information to proceed can be crucial. However, an agent that asks for confirmation on every step or that spends minutes at a time analyzing possible action sequences is not usable or effective. Thus, the final work item in this proposal is to develop a methodology for integrating the API validation and reasoning tools into the agent's training process. This will include developing a baseline for using direct reinforcement learning algorithms with tools [7, 12, 29] as well as exploring how to use the validation tools as a means for providing immediate feedback to the agent during training – allowing us to generate action rewards without a full trajectory and potentially drastically improving the process.

With an appropriate task set in place we will then proceed to investigate the integration of validation tooling directly into reinforcement learning algorithms. This will include exploring the use of reward shaping to drive the best use of validation tooling and immediate reward feedback from the validation tools to guide the agent's learning process toward quick conclusions instead of requiring extensive exploration.

Thus, the third work item in this proposal is to develop a framework for a fully integrated agentic API and tooling learning system. Our hypothesis is that the complimentary nature of the API specification, validation, and learning tools will allow us to create a system that is able to rapidly learn to accomplish tasks, adapt its behavior, and improve its performance over time.

In this theme our goal is to evaluate the impact of API specifications and validation tools on agent performance as well as exploring approaches for integrating the API features and tooling directly into the training phase of an agent. The first work item is to develop a baseline set of tasks and performance metrics to evaluate the impact of various tools on the agent's ability to complete tasks. Using this baseline we will then explore the impacts of API specifications and the agent's ability to reason about and use APIs. From there we will explore the potential for integrating the API validation and reasoning tools into the agent's training process.

3.3.1 Evaluation Suite and Baseline Performance.

To evaluate the impact of API specifications and validation tools on agent performance, we will develop a baseline set of tasks and performance metrics. Our first set of tasks will be based on the AppWorld [33] task set, which provides a rich set of tasks that require the agent to interact with a variety of APIs and services. Translating these APIs from their current form, text documentation and JSON schemas, into the Bosque API specs will provide experience with our specification language. We will also develop a set of tasks based on shell activities and scripting [15]. This second dataset represents a high-value domain for agents, has a different distribution of API behavior, and critically for us is a domain where incremental and interactive task completion is an appealing workflow. These datasets will provide an initial evaluation of the impacts of API design and access to tooling on the agentic performance.

Based on these evaluation suites we plan to conduct several baseline experiments. The first is to evaluate the impact of API specifications on agent performance *vs.* performance reported on the tasks using only textual documentation and JSON schemas. This will quantify the impact of API design on agent performance and which features of Bosque contribute the most to improved performance. The second baseline evaluation is to evaluate the impact of validation tools on agent performance. This will be based on a direct generate and check approach to task completion, where the agent generates multiple complete sequences of actions which are then filtered by the validation tools.

Task 3.1: Baseline Evaluation. *The objective of Task 3.1 is to develop a robust evaluation suite and baseline performance metrics for API and validation contributions of this research.* The results of this capability provide a foundation for the rest of the project and also represent a useful contribution to the community. High-quality evaluation suites across a variety of tasks and domains are a critical component of agentic research.

3.3.2 Integrating API Validation and Reasoning Tools into Agent Training.

The second work item in this theme is to develop a methodology for integrating API validation and reasoning tools into the agent's reinforcement learning training process. This will include developing a baseline for using direct reinforcement learning algorithms with tools [7, 12, 29, 33] as well as exploring how to use the validation tools as a means for providing immediate feedback to the agent during training.

Our first planned use of the validation tools is to provide the ability to terminate a rollout early if the current plan prefix is invalid. This will allow us to generate action rewards without a full trajectory. Using the Bosque static validator, we can determine if a call will fail or violate a policy and what actions were missed. Interestingly, we can also identify cases where where an API call fails due to an unsatisfied precondition and what actions were missed as in Figure 10. Thus, we plan to explore pseudo rewards even for actions that do not appear in a given trace with the goal of decreasing the number of needed learning steps and improving the agent's ability to learn from its mistakes.

The second area we plan to investigate is the use of the validator, and other static analysis tools, to provide more effective reward signals in general. Even among successful plans, the agent may make redundant calls or be overly eager in asking for user confirmation. We can use the validation tools, as well as other standard compiler techniques, to evaluate completed scripts and identify undesirable features to shape the reward function.

Task 3.2: Integration of API Validation and Reasoning Tools into Agent Training. *The objective of Task 3.2 is to develop a methodology for integrating API validation and reasoning tools into the agent's reinforcement learning training process.* The core techniques used in this task have been previously developed in the context of exposing other code generation tools, like semantic

analysis, type-checking, linting [14, 25, 28] to agents. However, given the substantial increase in power that the BOSQUE static validator provides, we expect to be able to achieve substantial improvements via the integration of these tools and enabling the agent to self-introspect on code it generates and actions it takes.

References

[1] M. Allamanis, E. T. Barr, C. Bird, P. T. Devanbu, M. Marron, and C. A. Sutton. Mining Semantic Loop Idioms. *IEEE Transactions on Software Engineering*, 2018.

[2] Amusuo, Paschal and Robinson, Kyle A. and Singla, Tanmay and Peng, Huiyun and Machiry, Aravind and Torres-Arias, Santiago and Simon, Laurent and Davis, James C. ZTD_{JAVA}: Mitigating Software Supply Chain Vulnerabilities via Zero-Trust Dependencies. ICSE, 2025.

[3] Anthropic Guide to Context Engineering. `https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/claude-4-best-practices`, 2025.

[4] Claude Code Best Practices. `https://www.anthropic.com/engineering/claude-code-best-practices`, 2025.

[5] V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful REST API fuzzing. ICSE, 2019.

[6] Bhargavi Paranjape and Scott Lundberg and Sameer Singh and Hannaneh Hajishirzi and Luke Zettlemoyer and Marco Tulio Ribeiro. ART: Automatic multi-step reasoning and tool-use for large language models, 2023.

[7] K. Chen, M. Cusumano-Towner, B. Huval, A. Petrenko, J. Hamburger, V. Koltun, and P. Krähenbühl. Reinforcement Learning for Long-Horizon Interactive LLM Agents, 2025.

[8] SMT-LIB Standard: Version 2.7. `https://smt-lib.org/papers/smt-lib-reference-v2.7-r2025-07-07.pdf`, 2025.

[9] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. Logics of Programs, 1982.

[10] Copilot Context Engineering. `https://code.visualstudio.com/docs/copilot/chat/copilot-chat-context`, 2024.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Language Systems*, 1991.

[12] J. Feng, S. Huang, X. Qu, G. Zhang, Y. Qin, B. Zhong, C. Jiang, J. Chi, and W. Zhong. ReTool: Reinforcement Learning for Strategic Tool Use in LLMs, 2025.

[13] R. T. Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, 2000.

[14] Copilot Programming Assistant. `https://github.com/features/copilot`, 2022.

[15] B. R. Gyawali, S. Achalla, K. Kallas, and S. Kumar. NaSh: Guardrails for an LLM-Powered Natural Language Shell, 2025.

[16] K. Havelund and G. Roşu. An Overview of the Runtime Verification Tool Java PathExplorer. In *RTS*, 2004.

[17] M. Kim, T. Stennett, S. Sinha, and A. Orso. A multi-agent approach for rest api testing with semantic graphs and llm-driven inputs, 2025. `https://arxiv.org/abs/2411.07098`.

[18] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally Specified Monitoring of Temporal Properties. In *Formal Methods in System Design*, 1999.

[19] Z3 SMT Theorem Prover. `https://github.com/Z3Prover/z3`, 2024.

[20] X. Ma, Y. Gong, P. He, H. Zhao, and N. Duan. Query Rewriting for Retrieval-Augmented Large Language Models, 2023.

[21] M. Marron. Programming Languages for AI Programing Agents (Invited Talk). DLS, 2023.

[22] M. Marron. Toward Programming Languages for Reasoning: Humans, Symbolic Systems, and AI Agents. Onward!, 2023.

[23] M. Marron. A Programming Language for Data and Configuration! Onward!, 2024.

[24] M. Marron. Small-Model Validation for Bosque Programs, 2025. Under Review.

[25] Model Context Protocol. `https://www.anthropic.com/news/model-context-protocol`, 2024.

[26] A. Pnueli. The temporal logic of programs. Foundations of Computer Science, 1977.

[27] Prompt Injection Attacks. `https://aws.amazon.com/blogs/security/safeguard-your-generative-ai-workloads-from-prompt-injections/`, 2025.

[28] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools, 2023.

[29] J. Singh, R. Magazine, Y. Pandya, and A. Nambi. Agentic Reasoning and Tool Integration for LLMs via Reinforcement Learning, 2025.

[30] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 1986.

[31] Tatsuro Inaba and Hirokazu Kiyomaru and Fei Cheng and Sadao Kurohashi. MultiTool-CoT: GPT-3 Can Use Multiple External Tools with Chain of Thought Prompting, 2023.

[32] G. Team. Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities, 2025.

[33] H. Trivedi, T. Khot, M. Hartmann, R. Manku, V. Dong, E. Li, S. Gupta, A. Sabharwal, and N. Balasubramanian. AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents. ACL, 2024.

[34] TypeSpec API Specification. `https://typespec.io/`, 2024.