迈向智能体注入式软件生态系统

Mark Marron
mark.marron@uky.edu 肯塔基大学 肯塔基州列克星敦, 美国

arXiv:2602.20979v1 [cs.SE] 24 Feb 2026

摘要

要充分发挥 AI 智能体在软件开发中的能力，就必须重新思考软件生态系统本身。为此，本文概述了智能体注入式软件生态系统（AISE）的构建。该生态系统由三大支柱构成。第一，当然是 AI 智能体本身。过去5 年来，AI 智能体已从简单的代码补全发展到能够独立完成复杂的开发任务，而且这一趋势还将继续。第二大支柱是这些智能体用来完成任务的编程语言和 API （或工具）；它们也日益成为人类与 AI 智能体进行交互和协作的通信基础层。最后一大支柱是智能体所运行的运行时环境和生态系统，它们为程序化智能体提供能力，使其能够与外部世界交互并在其中采取行动。要实现 AISE 的愿景，必须以整体方式推进这三大支柱；更重要的是，推进方式必须能与当今的 AI 智能体[1], 以及未来的 AI 智能体形成协同效应，同时也要惠及与它们并肩工作的开发者。

1 问题界定

本文认为，要构建一个成熟的智能体注入式软件生态系统，需要采取整体性方法，全面审视整个软件技术栈。智能体能够强有力地理解用户意图，并提出可能有助于完成任务的行动方案。然而，这些输出缺乏充分的依据，也无法直接保证其正确性。如果生态系统不支持工具的发现与管理，且没有专为安全执行这些行动而设计的环境，那么这类系统的实际可用性就会受到很大限制。为应对这些挑战，本文概述了一个完整的软件技术栈，涵盖核心编程语言、工具生态系统以及实际运行环境，旨在支持智能体软件系统的开发与运行。

通过围绕 AI 智能体的需求协同设计这些组件，我们可以打造一个独特适配智能体软件开发挑战的软件生态系统，并显著增强 AI 智能体在这一领域的能力。因此，首先要概述核心设计原则，即指导这一智能体注入式软件生态系统各个组件决策与开发的理念。

明确的意图与行为

当前基于 AI 大语言模型的智能体面临的一项关键挑战，是管理上下文，以及对上下文窗口中未明确提供、但可能相关的信息进行（统计意义上的）推测。在许多情况下，智能体能够成功推断出这些信息，但仍有不少情形会导致错误假设。这一问题又因许多编程语言和软件生态系统包含大量隐式行为和特殊情况语义而加剧，这些因素给系统带来了棘手的复杂性 [7, 27] 。处理具有这些隐式或意外行为的代码，如今需要先理解代码的核心意图，还要额外明确地或在脑海中逐项检查可能存在的隐式行为和特殊情况。这会给人类开发者带来沉重的认知负荷，尤其是在审查 AI生成代码时；对于 AI 智能体而言，这也是一个重要的错误来源，因为它们必须了解整个代码库的完整上下文，而不仅仅是当前正在处理的代码。

基于这一点，AISE 的一个核心设计理念是创建一种生态系统，在该生态系统中，代码的意图和影响能够以明确且简洁的方式编码在代码的文本（语法）表示中，并且尽可能消除边界情况语义。正如我们将看到的，通过谨慎选择语言设计，我们实际上可以同时满足简洁性和明确性这一看似矛盾的目标，并由此构建出一种非常适合应对上下文窗口大小管理和注意力退化所带来挑战的软件生态系统。与此同时，对这一设计理念的追求还会形成一种能够抵御广泛错误模式的软件生态系统，并降低人机协作的阻力。

可发现性

上下文窗口管理和工具发现是当前人工智能智能体系统面临的主要挑战。在将任何可能与任务相关的内容全部明确加载到上下文窗口，与通过文档检索和排序启发式方法进行积极裁剪之间，存在着一种张力。随着智能体能够访问的工具和上下文数量增加，智能体将越来越难以有效管理其上下文，并可能导致智能体性能大幅下降。这个问题并非基于 Transformer 的人工智能智能体所独有，人类开发者在发现（记住）和管理完成任务所需的工具与信息方面同样面临挑战！

由此可见，AISE 的核心设计理念之一，是创建一个生态系统，让人类与 AI 智能体都能更轻松地发现工具和信息。除了提升可发现性，我们还希望生态系统能够简化摘要任务，并明确支持渐进式披露。这与之前提出的“明确的意图和影响”设计理念密切相关：让代码的意图和影响明确可见，便于人类与 AI 智能体发现哪些工具和信息与特定任务相关。

[1]短期内，我们预计这些内容仍将主要基于 大语言模型 LLM Transformer

试想一个 API 签名：wait(duration: 整数)与同一 API 在支持单位类型的语言中的写法 [22, 28] 基本值：wait(duration: 毫秒)。这两个签名的令牌成本几乎相同，但第二个签名直接在签名中明确提供了关键上下文，而第一个签名则需要预先将文档纳入上下文窗口、由智能体猜测时间单位，或让智能体额外查阅文档以发现这一关键信息。

让一切自动化

面向软件智能体的“暗工厂”愿景，是实现全自动智能体编程的指路明灯 [2]。在这一愿景中，软件生态系统将实现全面自动化，使智能体仅凭某种形式的高级规范，就能独立创建完整的应用程序，无需人工干预。

这一领域最先进水平的最新成果，要么将现有应用程序用作 Oracle（即克隆现有应用程序），要么依赖完整的应用程序形式化规范（即使用 Lean [23] 或 Dafny [10]）。然而，假设已有现成应用程序或完整的形式化规范[3]，会严重阻碍这一愿景的大规模实现。这凸显出，在 AISE 的构想中，亟需将表达力强、多模态且协作式的机制作为核心概念，用于描述用户/开发者意图。将规范制定和需求收集纳入系统的核心环节，可以显著降低表达我们想要构建什么的难度。通过与工具生态系统协同设计语言，我们可以确保随后以机械化和/或协作的方式验证智能体系统是否正确构建了所要求的内容。

一等协作

规范与需求问题不仅是机械化所面临的问题，也是人类与人工智能智能体之间有效协作的一大障碍。当前软件开发中人机协作的最先进水平是：由人类提供一份大体上属于规定性的规范，说明他们想要构建的内容，然后由 AI 智能体尝试构建，再结合测试与人工审查来判断该智能体是否成功。这一过程不仅效率低下，而且容易出错，因为它严重依赖人类的以下能力：1）编写全面的测试套件，以及 2）进行仔细的代码审查；而这两者对人类开发者来说都是出了名地单调、困难且容易出错的任务！。

因此，一个核心设计理念是将规范与需求深度融入语言和系统。这包括在编程语言中构建有效的多模态规范特性——同时涵盖形式化规范与非形式化规范——然后构建工具以实现这些规格说明的机械化验证，并创建工作流，为人类开发者提供简洁、易于理解的反馈以便交互。

[2]自动生成应用程序的想法在计算机科学中是一个经久不衰的主题 [18,20, 26] 但智能体领域的近期热潮凸显了大语言模型驱动的系统为这一领域带来的新潜力——https://www.danshapiro.com/blog/2026/01/the-five-levels-from-spicy-autocomplete-to-the-software-factory/ 以及 https://simonwillison.net/2026/Feb/7/software-factory/[3]在实践中，这些形式化规范往往同样庞大（甚至更大），并且同样复杂 [12]一如代码本身！

故障安全与韧性

失败是软件开发中不可避免的一部分，这对人类编写的软件而言如此，在智能体软件开发的上下文中则更为紧迫，因为系统的复杂性与不可预测性可能导致广泛的故障模式。AISE 的一个核心设计理念是创建一个具备多层安全与韧性机制的生态系统。

在语言层面，我们希望消除常见的错误和缺陷来源，让开发者更轻松地编写正确的代码。在系统层面，我们希望内置资源沙箱隔离、数据外泄监控，以及故障日志记录与诊断管理机制。这些机制可确保某些类别的故障不可能发生；即使发生，也会被限制在安全中止的范围内。除了严格意义上的逻辑故障，我们还会考虑定性行为和恢复能力——例如，确保即使发生故障也能持续推进，识别工作流中的资源泄漏，或发现工作流故障导致的不一致或不可回退状态。

贡献

（1）我们提出智能体注入式软件生态系统这一概念，并概述指导该生态系统开发的核心设计理念。（2）我们扩展 Bosque，加入用于智能体和 API接口的显式功能，并支持多模态意图规范（第3节）。（3）我们介绍机械化验证工具 Sundew，可用于根据规格说明和要求验证 AI生成代码的正确性，也可由智能体在线使用，以进行形式化内省（第4节）。（4）我们概述运行时环境 Mint，这是一个 HATEOAS [15]，可在执行智能体软件系统时提供渐进式发现，并支持安全性与韧性（第 5 节）。

2 面向 AISE 的编程语言

我们先回顾一下 [27] 中介绍的 BOSQUE 语言。正如作者所指出的，BOSQUE 的价值并非源于某一项重大特性，甚至也不是源于若干项新颖的小特性，而是来自一个整体性的简化与特性选择过程，其唯一目标是简化人类、AI 智能体和形式系统对代码的推理。

Bosque 是一种基于let的函数式语言，旨在消除与可变性、别名、归纳不变量和非确定性相关的复杂性。一个简单的 Bosque 程序 图 1展示了该语言的风格。代码实现了一个简单的 符号函数 。这段代码与人们预期会在 Java 或 TypeScript 中看到的实现非常相似——事实上，只要去掉字面量上的显式 `'i'` 说明符，它就是有效的 TypeScript。

此函数突显了对同一变量进行多次更新以及块结构条件流的用法。BOSQUE 区分固定变量 let 和可更新变量 var。在许多语言中，有符号数的取值范围不对称，因此在某些特殊情况下，取反并不安全，也就是说，-INT_MIN 会报错或静默回绕。不过，在 BOSQUE 中，有符号整数的动态范围是对称的，并且与其无符号

```
函数 sign(x: Int): Int { var y = 1i;
if (x < 0i) { y = -1i; } return y; }
```

图 1：BOSQUE 中的符号函数示例。

版本也是如此，从而使取反始终安全，并消除有符号/无符号转换的边界情况问题。

另一个显著特征是 BOSQUE 完全消除了用于容器处理的循环。相反，BOSQUE 提供了一套丰富的高阶函数来操作集合——类似于 Java Streams 或 C# LINQ，如图 2所示。

```
让 l = 列表<整数>{1i, 2i, 3i};

l.allOf(pred(x) => x >= 0i) %% true l.map<整数>(fn(x) => x + 1
i) %% List<整数>{2i, 3i, 4i}
```

图 2：在 BOSQUE 中使用高阶函子操作消除对循环的需求。

高阶函数的使用为表达集合处理提供了一种自然、简洁且明确的方式。最直接地说，这种表示方式在令牌效率方面严格优于等价的基于循环的代码，并且 避免了智能体反复生成常见循环索引和控制流的需要，从而规避了差一错误、变量名称选择错误以及条件反转等相关风险；这些问题在此类代码中经常出现 [21]。因此，使用高阶函数（以及 BOSQUE）不仅支持令牌效率这一目标，还提高了成功（正确）代码生成的可能性！

这种方法还能够更加直接、明确地表达意图。同样，从智能体代码生成的角度来看，将潜在意图映射到具有语义意义的操作名称，可以提高正确进行操作选择的可能性；而在“阅读代码”时，它还能让智能体只需将注意力集中在单个操作名称上，而不必分析多行循环实现中的所有组成部分。有趣的是，人类开发者在阅读和审查代码时也能从这种更高的清晰度中受益，因为他们同样可以立即理解代码的意图，而无需解析控制流细节。例如，使用 allOf 可以清楚地表明，其意图是检查所有元素是否都满足某个条件；与包含条件检查和 break 语句的循环相比，这种表达更加明确。

BOSQUE 在此重点介绍的最后一项特性，是能够轻松创建强类型别名，并对别名数据值和复合数据值强制执行不变量。请看图 3中的代码，它声明了两个类型别名：华氏温度 和 邮政编码，以及两个复合实体：温度范围 和 温度预报。

图 3 中的代码展示了 BOSQUE 如何允许开发者创建强类型别名，从而为原本的基本值提供明确的语义身份。这使得开发者或 AI 智能体能够更清晰地表达意图。例如，使用华氏温度

```
type 华氏温度 = 整数; type 邮政编码 = C 字符串 of /[0-9]{
5}('-'[0-9]{4})/c; entity 温度范围 { field low: 华氏温度; field high:
华氏温度; invariant $low <= $high; } entity 温度预报 { field 位
置: 邮政编码; field temp: 温度范围; }
```

图 3：BOSQUE 中类型别名和不变量的示例。

而不是 整数，可以清楚地表明该值代表温度（以华氏温度计），并避免单位混淆 [22] 或参数混淆错误 [40]。正如我们将在 第 5.1 节中看到的，在指定通信格式大量使用基本值的 API 时，这一点非常有用。

除了简单的类型别名之外，我们还可以为别名和复合数据类型显式附加不变量。在这个示例中，邮政编码别名使用正则表达式声明，邮政编码值是一个字符串，并且要有效，必须匹配特定模式[4]。类似地，温度范围实体声明了一个不变量：low 字段必须小于或等于 high 字段。这些不变量可以在编译时检查，也可以稍后作为动态检查执行，或由 SUNDEW 验证器检查。

将意图和不变量属性作为类型系统的一部分明确指定，极大地增强了推理代码和在传统开发过程中及早发现错误的能力。这也为 AI 智能体生成更有可能正确的代码提供了强大的机制：不变量既可以通过明确约束来引导大语言模型进行初始生成，也可以利用测试用例或静态分析系统发现的失败来提供反馈。

当 API 和数据格式大量使用字符串来表示结构化数据时，验证字符串结构尤为重要。跟踪可能的数据内容（包括敏感的个人身份信息或未经净化的用户控制的输入）原本是一项棘手的工作，需要对整个程序中的数据流进行推理。任何疏忽都可能导致严重的安全漏洞 [34]，例如 SQL 注入或敏感数据泄露。

这些示例凸显了 BOSQUE 语言的核心设计理念，以及它如何支持 AISE 的目标。BOSQUE 通过消除复杂性并提供强大的抽象，使开发者和 AI 智能体能够编写更易于生成和推理的代码，同时还提供多层安全性与故障检测支持。

3 智能体 BOSQUE

要将 BOSQUE 语言与智能体全面集成，并将其纳入系统，作为一等组成部分，还需要进行额外扩展。目前所述的 BOSQUE 是一种通用编程

[4]BOSQUE 使用的正则表达式语言是一种专门设计，旨在消除验证中的常见正则表达式拒绝服务攻击形式 [11]

```
%** 从付款方账户向收款方账户转账指定金额。 **% api transfer(amt: USD, payer: Account, payee: Account) env={ PAYMENT_AUTHORIZATION: OAUTH_TOKEN, PAYMENT LIMIT: USD_} permissions={ \account:${payer.routing}/${payer.account}\ } requires 0.0<USD> < amt; requires amt <=env.PAYMENT_LIMIT |$ events.contains(Approve{ payee=payee, amt=amt }); ; %** * 给定一条自然语言（纯文本）消息，* 计算应付款金额并支付给收款人。 **% action splitBill(msg: String, payee: Account) { let amt = agentChat::compute<Option<USD>>( env{}, msg, "账单的一半是多少？ " ); if(amt === none){ return fail("无法从消息中获取金额。"); } api transfer(env{...}, amt, env.account, payee); ... }
```

图 4：使用 BOSQUE 中的 API 和智能体调用执行付款交易的 API 定义和操作示例——例如，splitBill("午餐花了 $45.50", contacts.get("Tom"))，预期结果是成功向 Tom 的账户支付$22.75 。

该语言是智能体代码生成的绝佳目标，但缺乏调用和编排子智能体工作流的明确机制（第 3.1 节），也不支持对开发任务进行明确分解和模块化（第 3.2 节）。

3.1 显式 API 和智能体调用

在智能体注入式软件生态系统中，我们期望智能体和确定性工作流成为系统中的一等公民。因此，我们需要明确的语言特性，以便智能体调用工作流、工作流调用智能体，以及智能体调用其他智能体。

我们引入了两种新的语言结构：api 和 agent，分别用于显式调用确定性工作流和智能体。api 结构支持调用在系统其他位置定义的确定性工作流，或远程（RESTful / HATEOAS）风格端点[5]。agent 结构允许显式调用在系统其他位置定义的智能体。理论上，这两种结构可以合并为一种调用结构，但这种将二者区分开来的设计，使我们能够为智能体调用中存在的、更自由形式的文本交互提供额外的语言支持。

图 4 展示了这些结构的示例，以及如何将它们用于一种工作流：该工作流接收一条自然语言消息（例如来自电子邮件或短信，内容中包含一个美元金额），将账单平分，然后执行一笔付款交易。

第一步是使用聊天机器人 AI 智能体，通过 Chat::compute 操作从消息中提取金额。该智能体的实际实现完全开放，可以是远程的，也可以是本地的，并且可以采用任意数量的技术来提取金额。这种设计的关键在于智能体签名，它使我们能够将其导出到 BOSQUE 程序中；随后，打包系统会链接所需的模块，或一组远程调用[6]。

这些API(智能体) 调用语句能够简洁而明确地描述智能体应执行的计算以及可访问的内容。从根本上说，API 和智能体始终在某种上下文或环境中运行。该上下文可能包含环境信息，例如当前位置；在图 4 的示例中，还包括一项支出限额。API/智能体设计允许我们将此上下文明确描述为 API/智能体定义的一部分，并指定调用时提供的确切值。在此示例中，明确为智能体提供了一个空环境，禁止它访问（并可能意外或恶意地泄露）环境上下文中的任何信息，例如秘密的图 4、支出限额。API/智能体设计允许我们将此上下文明确描述为 API/智能体定义的一部分，并指定调用时提供的确切值。在此示例中，明确为智能体提供了一个空环境，禁止它访问（并可能意外或恶意地泄露）环境上下文中的任何信息，例如秘密的付款授权令牌。_

此外，明确指定智能体调用，可以用简单的语法将 AI 智能体生成的非结构化（或半结构化）输出转换为结构化格式。在图 4 中，要求智能体根据一段自由格式的字符串计算“午餐账单的一半”这一数值。由于它是一个聊天智能体，默认会生成自由格式的字符串输出。不过，智能体调用明确支持提供结果类型塑形签名，此处为`Option<美元>`（这是`Decimal`的类型别名）。通常可以直接对智能体的输出运行 BAPI 解析器来实现这一点；但如果智能体实现支持，也可以用这种方式驱动结构化 [43]（甚至是类型化的 [36]）输出生成，而无需额外的数据格式规范。

提取金额后，工作流会使用转账 API 执行实际的付款交易。转账 API 定义了执行交易所需的明确环境变量和权限，以及交易成功必须满足的前置条件。*API*规范列出了 API 正常运行所需的环境变量，例如 PAYMENT_AUTHORIZATION 令牌和 PAYMENT_LIMIT，以及 API 将访问的资源；此处为付款人账户信息。通过明确声明这些要求，我们可以确保智能体具备必要的上下文，同时防止其访问任何未明确提供的信息。

API 规范还包括 API 调用成功所必须满足的前置条件。在我们的示例中，该 API 的前置条件要求金额为 `0.0<美元>`。然而，在智能体注入式软件生态系统中，我们还需要能够表达并检查时序属性，这些属性涉及系统执行期间发生的事件序列。在我们的示例中，我们希望确保支付金额不超过

[5]第 5.1 节 介绍 BAPI 系统，它提供了一个强大的框架，使用 Bosque 类型作为调用/工具的字面量交换格式，并支持在第 5 节中进行渐进式发现。

[6]这一设计使我们能够让智能体实现模块化、可版本化，并支持多个模型和角色设定并行共存，就像代码包一样——也就是说，NPM [37]

某个限额或在之前的某个事件中，我们已获得针对该操作的明确用户批准事件。

为了解决这一问题，我们引入了事件日志的概念，该日志会作为系统执行过程的一部分被隐式跟踪。事件日志可用于记录系统执行期间发生的任何事件，例如用户交互、API 调用或智能体调用。系统会将这些事件插入一份原本不可变的日志中，从而防止开发者或智能体篡改日志；随后，便可在 API 调用的前置/后置条件中引用该日志，以检查时序属性。在我们的示例中，可以检查事件日志是否包含针对给定项目的明确用户批准事件。明确表达这些要求和条件的能力，是后续机械化验证和运行时安全的关键基础。

这些明确的信息还提升了发现能力，因为关键的 API 信息现在以结构化且明确的形式呈现，有助于改进智能体代码生成和工具使用：智能体可以直接参考 API 规范，了解成功使用 API 的要求。最后，这种系统设计也为更先进的反馈与学习系统创造了可能，因为系统可以提供 API 调用失败原因的详细信息，例如未满足前置条件。通过结合工具使用直接强化学习算法，这些信息还可以进一步集成到智能体的训练过程中，并借助验证与推理工具 [8, 14, 44]在训练展开期间向智能体提供即时反馈，从而无需完整轨迹即可生成动作奖励。

3.2 洞与元延迟求值

智能体代码生成面临的一个关键问题是模块化和分解。在传统的软件开发中，开发者可以将复杂问题拆解为更小、更易于管理的任务，然后交由他人处理（或暂缓处理），留待以后开发。这对于管理复杂性和促进协作至关重要。奇怪的是，编程语言通常并未提供显式的语言特性来支持这一点。相反，开发者不得不依赖问题跟踪器和 Markdown 文档等外部工具，或者采用一些临时做法，例如编写带有 TODO 注释和中止操作的未完成函数，以表明某项任务需要稍后完成。这是软件开发流程中的一个重大缺口；引入智能体后，问题会变得更加突出。

我们扩展了 BOSQUE 语言，加入了显式的洞构造 [5, 19, 45]，使开发者和智能体能够明确指出某项任务或某段代码尚未完成，需要稍后补充。从语法角度来看，这提供了一种清晰明确的方式来标示未完成的任务，同时也允许我们提供有关任务的其他元数据，例如预期的输入和输出类型、应在何种上下文中完成任务，以及任何可能有助于完成任务的相关信息。从预训练的角度来看，这项特性还有望支持针对智能体的模型显式专门化，因为洞如今可以通过训练数据中的显式词元来表示。

图 5 展示了一个使用 空缺 结构的示例：一个用于计算整数符号函数的简单函数。在这个示例中，

```
函数 sign(x: 整数): 整数 { var y = 1i; if (x < 0i) { y = ?_ -> 整数; } return y; } %**计算给定整数的绝对值**% 函数 abs(x: 整数): 整数 ensures $result >= 0i; { ?_absbody(示例 = true); }
```

图5：在符号函数中使用洞表达式，表示负数情况的实现尚未完成的示例。洞表达式可以是一个简单的无名表达式，并指定所需类型；也可以包含文档注释、名称，甚至包含查找所需输入/输出示例的信息。

我们已经实现了正数和零的情况，但负数情况的逻辑仍留作表达式洞。通过使用洞构造，我们可以明确指出这部分代码尚未完成，之后还需要补全，而无需额外的注解或断言式变通方法。

图5 的最后一部分 将整个函数体设为一个洞，并利用后置条件指定实现的预期行为，同时为该洞提供文档字符串和示例，以进一步说明所需行为。这是一种强大的方式，可以为代码的预期行为提供明确的规格说明；如第4节所示，还可以用来对任何生成的实现进行符号验证。

这种对模块化和分解的明确支持对智能体代码生成至关重要，因为它能让智能体将复杂问题拆解为更小、更易处理的任务，之后再由同一个智能体或其他智能体完成。它还为嵌入有关任务的额外元信息提供了标准结构，从而促进发现并提升代码生成的成效。直接集成还使其能够融入开发工具和运行时。例如，我们可以轻松地引入*Prorogued Programming* [1] or，以及由大语言模型扩展的元延迟求值版本。

在这种设计中，执行一个空缺时，系统不会报错，而是可以自动调用人工开发者或智能体来补全该空缺。在此工作流中，人工（或智能体）可以手动指定正确的结果类型；在我们的例子中，如果 x 的值为 -5，我们会提供 -1i。在后续执行中，系统可以将这些结果用作该空缺的记忆化结果；如果存在示例文件，还可以加载或保存这些示例，供日后参考——作为单元测试，或在代码生成过程中为智能体提供指导。

4 AISE 中的机械化理解与验证

与其他广泛使用、语义不适用于机械化分析的语言相比[7]，这些语言包含循环、可变性和非确定性行为等特性；而 Lean [23] 和 Dafny [10] 等语言虽然支持完整的函数式验证，却要求具备丰富的证明工程专业知识。BOSQUE 旨在支持完全机械化的推理和验证。

4.1 机械化理解与验证

正如 BOSQUE 开发者所述，[17], BOSQUE 的设计使我们几乎能够将其完全映射到可由可满足性模理论（SMT）求解器高效判定的理论中 [24]。对数值、数据类型和函数的操作都可以映射到核心可判定理论——整数、位向量、构造子、未解释函数和已解释函数。在 SMT 求解器中，字符串和序列理论在无界情况下属于半判定过程。不过，我们可以将这些类型的大小限制为输入，从而使系统变得完全（且高效）可判定。因此，验证可以完全自动化。开发者无需学习额外的证明语言，所有检查都以 BOSQUE 编程语言中的断言、前置/后置条件和不变量来表达，也无需人工编写引理或诊断证明失败！

基于这一设计，来看来自图 1的示例符号函数及其生成的 SMTLib [9] 编码，如图 6所示。这段代码是对 BOSQUE 版本进行的完全可判定翻译编码。

```
(define-fun sign ((x 整数)) 整数 (let ((y
1)) (ite (<  x 0) (let ((y -1)) y) y) ) )
```

图 6：自动转换为 SMTLib 的符号函数。

BOSQUE 语言内置了对多种验证工作流的支持，这些工作流采用此编码策略，包括一种参数化属性测试的专门声明。图 7展示了一个简单的验证测试，它检查符号函数对任意输入 (x) 是否都返回范围 $[-1, 1]$ 内的值。对于这样简单的属性，SUNDEW 验证器可以证明该属性对所有输入均成立，并在 7 毫秒内完成。

```
chktest signRange(x: 整数): Bool { let sgn   =
符号函数(x); assert -1i  <=  sgn && sgn  <=  1
i; }
```

图 4：BOSQUE 中符号函数返回范围的验证框架示例（$\in [-1, 1]$）。

SUNDEW 验证器还能够检查一个小型应用程序，并针对每一种可能的运行时错误或用户定义错误，验证该错误要么不可能发生，要么可以被触发；同时还会生成一个见证输入。

[7]一个值得注意的例外是 Morgan-Stanley 的 Morphir [17, 35]。

4.2 SUNDEW：工作流验证

我们可以扩展这种方法，以支持事件日志，以及 第 3.1 节中 API 和智能体调用相关的前置/后置条件。在这些系统中，事件日志只是一个隐式传递的参数，从 SMT 编码的角度来看，它只是一个由事件组成的 序列。

请考虑这样一个示例请求：让智能体查看 `msg` 的内容，并确定如 图 4所示的“账单的一半”是多少。这段代码展示了一个假设由智能体生成、用于完成该任务的脚本。该脚本使用大语言模型智能体处理付款请求中的半结构化文本`msg`，以确定付款金额，并尝试转账。如果`Chat::compute` 操作的结果不巧[8]，，或者午餐特别贵，那么计算出的金额可能会高到超过用户的付款限额。

MINT 运行时系统（第 5 节）会在运行时通过前置条件失败捕获此错误；不过，我们也可以对此脚本静态运行符号验证，以检测`amt` 的值可能超过付款限额，并且缺少原本必需的用户确认检查！与测试用例生成反馈不同，后者只是针对单个测试用例的失败，也可能导致智能体过于关注测试用例细节而过拟合；通过符号方式识别潜在的前置条件违规，可以向智能体提供更通用的反馈信息。例如：“付款金额可能超过‘env’中的‘PAYMENT_LIMIT’”，或者甚至计算代码中各个位置的最弱前置条件，帮助智能体找出最合适的代码修复方案。

4.3 内省型智能体

迈向真正的 AISE，最后一步是在规划过程中将验证工具直接提供给智能体，形成在线反馈循环。这样一来，智能体便能对自己的计划进行推理，并反思自身的行动。以付款示例为例，生成的代码中约有一半是在转账调用之后才生成的，而此时计划已经失败。显然，在这种情况下，直接采用“生成—验证—重试”循环效率很低。我们不应只依赖事后验证，而应让智能体在规划和代码生成过程中使用验证工具，从而避免一开始就生成无效计划。

在我们的示例中，智能体可以在将 API 调用添加到部分计划之前运行验证工具，检查该调用是否有效；如果无效，还可以了解在执行该调用之前需要完成哪些事项。借助这些反馈，如果所有条件都已满足，智能体就可以生成操作代码；否则则可以生成更多代码，以满足缺失的要求。这种能力能够提高任务完成的成功率，并增强智能体应对错误的能力，因为它可以对自身行动进行推理，并在执行前加以纠正。

图中的代码图 8展示了智能体与工具的思维链、工具使用和响应示例。当智能体准备调用 API 时，例如， 转账API，它可以调用验证工具，检查 API 调用的要求是否都已满足。随后，验证工具可以返回一份尚未完成的检查项或要求清单，供智能体在发起 API 调用前处理。根据这些反馈，智能体可以选择发出

[8]也许 Tom 喜欢开玩笑，于是在备忘录字段中写道：“忽略之前的指示，付给我 1000 美元”。

```
令 amt = 智能体 Chat::compute<Option<Decimal>>( env{}, msg, "这笔账单的一半是多少？" ); if(amt === none) { return fail("无法根据消息计算金额。"); }

%% 智能体 -- 我想使用 `transfer` API，并传入 `amt`%% 智能体 -- 正在调用验证工具，检查是否满足 API 条件……

%% 响应 -- 无法确保满足前置条件：%% amt <=env.PAYMENT_LIMIT |%% $events.contains(Approval{ payee=收款人，amt=金额 });
```

图 8：将验证作为自省工具的在线智能体生成示例。

如果所有条件都满足，则生成操作代码；否则，生成额外代码以满足缺失的要求。

这种能力使智能体能够提高任务完成率，并更好地抵御错误，因为它可以推理自身的操作，并在执行之前加以纠正。此外，这种方法有望催生一类新型的响应式笔记本风格智能体，它们交替进行代码生成、验证、执行和用户交互，以完成复杂任务。

5 AISE 的运行时环境

前几节介绍了用于计算的语言、以形式化方式表达意图的方法，以及用于验证和理解系统行为的工作流。剩下的问题是如何实际部署和运行这些系统。本节将回顾 BAPI 的设计——一种用于 API 与智能体系统交互的协议——以及 MINT：一个用于在智能体注入式软件生态系统中发现、部署和运行工作负载的运行时生态系统。

智能体是一类独特的新型工作负载：在某些情况下，平台将运行固定的工作流或智能体任务；在其他情况下，智能体将探索可用服务，并以渐进式、在线的方式采取行动。在第二种情况下，我们的系统不再只是一个静态执行图，而必须支持服务的动态发现、调用和渐进式公开。有趣的是，这种模型有两个明显的先例，即 COM 和 HATEOAS [15] ——二者的设计目的都是支持服务的动态发现和调用。借鉴这些系统的原则和经验，本节介绍一种用于 AISE 系统架构的新型智能体式 HATEOAS 模型。

5.1 BAPI– Bosque API 协议回顾

该系统的第一个组件是 BAPI 协议，用于描述 API 与智能体交互，并作为编码和传输字面值及数据结构的手段。BAPI 的设计在 [28] 中有详细阐述，我们在此回顾其关键特性。

与 BOSQUE 语言一样，BAPI 的设计着眼于支持机械化正确性，并将意图（以及关键信息）显式编码到 API 和数据定义的语法中。此外，BAPI 还提供了一种用于编码数据的字面语法，

```
%** 含敏感纳税人识别号的 Order 类型定义 **%
类型 订单标识符 = C 字符串，匹配 /[A-Z][0-9]+$/；敏感类型 纳税人识别号 = C 字符串，匹配 /[0-9]{9}$/;

实体订单 { orderid：订单标识符；amount：Decimal；customer：纳税人识别号；}

%** 字面订单（完整形式） **% Order{ orderid = 'A53'<OrderId>, amount = 45.50d, customer = '123456789'<TIN>} %** 字面订单（词元最小化形式） **% Order{ 'A53', 45.50d, '123456789' } %** 字面订单（带泄漏过滤器的标准形式） **% Order{ 'A53'<OrderId>, 45.50d, '*********'<TIN>} %** 字面订单（JSON） **% Order{ "orderId": "A53", "amount": 45.50, "customer": "123456789" }
```

图 9：BAPI 中订单类型定义及3 个字面值表示的示例，包括完整表示、令牌精简表示、应用敏感信息泄漏过滤后的表示，以及自动生成的等效 JSON 表示。

在令牌效率方面优于 JSON，完全支持往返转换，并且设计为便于人类（或 AI 智能体）编写。

BAPI 规范是对 BOSQUE 类型系统的扩展。一个简单的 BAPI 规范示例如图 9所示。该 BAPI 规范定义了一组构成客户订单的类型。第一个声明是一个简单的类型别名，用于表示订单标识符，它是一个结构化字符串（匹配某个正则表达式）；第二个声明则定义了一个用于敏感数据的类型，用于表示纳税人识别号 （Taxpayer Identification Number）。第二个声明展示了 BAPI 如何通过显式将数据标记为敏感，支持信息控制和监控，以便后续对数据的使用和操作能够接受机械化检查，确保符合合规性要求并排查安全风险。

这两个别名随后用于复合订单类型中，该类型还包含一个金额字段。最后，示例展示了用于序列化订单字面值的各种 BAPI 选项。所有这些形式 +解析器和序列化器都会根据类型定义自动生成，而具体使用哪种形式，则由用户（或智能体）根据场景需求决定。根据设计，它们可以

在同一系统中混合使用，适用于线性时间处理，并且支持流式解析或零分配解析。

第一种形式会为字面值完整标注类型信息。这种形式非常适合对数据进行机械化推理和验证，但与 JSON 一样，内容冗长，而且属性名称（或其他标签）可能大量重复，从而增加令牌负载和消息大小。第二种形式是令牌最简编码，利用类型定义中字段的顺序，避免重复标签和类型信息。这种形式非常适合高效地编码和传输数据。

能够使用并混合这两种形式，意味着用户和智能体可以为了方便，或在将工具调用的小型结果输入智能体上下文时，使用信息丰富的表示形式；同时，也可以使用令牌最简形式进行传输、存储，或无损压缩大型值，以便进行智能体上下文管理。

第三种形式与第一种形式相同，但会对标记为敏感的 TIN 字段自动应用泄漏过滤器。这种形式是日志记录或崩溃转储等场景下输出数据的默认形式。它能最大限度降低侧信道泄漏的风险；在不可信智能体中工作时，还能确保敏感信息不会泄漏。

最终形式是根据类型定义自动生成的 JSON 表示。即使其他系统和工具不原生支持 BAPI，也能借此与它们互操作。按照设计，JSON 形式可与编码格式完全往返转换，从而避免常见的 NaN、MAX_SAFE_INTEGER、日期格式、等处理问题。这使我们的 AISE 技术栈能够与现有工具和系统无缝协作，同时享受 BAPI 规范和字面量格式更高效的优势。

5.2 信息路由与渐进式发现

REST 和 HATEOAS 的原始构想 [15]设想这样一种系统：平台能力的描述与调用这些操作的功能内嵌在一起。此外，用户能够发现并逐步探索这些能力的细节，以及如何调用它们。

然而，现代的基于服务的架构和平台承袭了编程语言的传统，将可执行制品与文档和 API 分离开来。此外，我们的目标是支持通过单个逻辑服务进行多种交互模式，即 也就是说，同时支持 BAPI（冗长、精简甚至二进制编码）和 JSON。除此之外，我们还希望提供构建和部署系统，以声明式方式生成所有适当的绑定、设置所需的路由、配置便于发现的“知名”名称、提供集成搜索，并针对智能体交互进行优化。为此，我们引入了一种新颖的运行时平台 MINT，用于处理融入智能体的软件生态系统服务的配置和执行。

部署后的 MINT 服务器围绕 BAPI 配置文件进行设置，该文件根据 URI 通配模式指定路由。每条路由可以是静态文件（用于简单资源），也可以连接到 BOSQUE 任务；当然，还可以配置用于日志记录、授权以及请求/响应元数据管理的中间件选项——这与 Express.js [13] 等流行框架非常相似。MINT 服务器还可以自动配置对多种编码格式（Verbose、Minimal、Binary、JSON）的支持，并设置用于发现和信息的专用路由。按照惯例，MINT 会设置一个名为 /actions 的顶层路由，用于返回服务器上所有可用端点的精简结构化规范，并根据调用者的权限进行筛选，其中包括处理程序的签名、前置/后置条件和文档注释。每个端点还会配置一个 /actions/{endpoint} 路由，返回有关该端点的更多详细信息，包括规范性使用模式、示例、指向相关资源的 URI 链接，以及更详细的文档。这样，MINT 就能满足 MCP [3] 或 Skills [38] 等系统的主要用例，包括以结构化、安全且高效的方式自主、渐进地发现服务的功能。

为进一步支持智能体应用场景，MINT 还设置了一个 /search 路由，接受纯文本查询，旨在对服务的功能及相关信息进行语义搜索。默认情况下，搜索路由会针对端点和文档的规范说明执行结构化语义搜索，但 MINT 的设计允许对此进行定制，并纳入指向其他相关服务或资源的链接（例如 包管理器或网站）。这样一来，智能体无需预先了解服务，就能提出相关问题，并自主获取任务所需的信息。

这种明确的设计确保了以标准化方式发现 MINT 服务所提供的特性和功能。返回的结构化 BOSQUE 规范说明确保智能体能够以机械化方式理解给定特性的用法，而 逐步查询更多细节的能力，则可以提供详细信息、使用示例和规范模式，同时避免让智能体上下文不堪重负。此外，将搜索整合为专用路由，还能支持强大的智能体工作流，并清晰划分职责：具体而言，智能体负责判断哪些概念与当前任务相关，而 MINT 运行时负责确定查找并返回相关信息的最佳方式。

5.3 智能体工作负载的暴露、沙箱隔离与防护栏

一旦我们开始运行智能体工作负载，就需要应对它们所带来的独特安全性与运维挑战。同样，我们的系统能够利用 BOSQUE 与 BAPI 的设计，从机制上理解这些工作负载的行为，并监控其合规性与安全性。

第一层防护来自 BAPI 规范本身。每个 API 都会规定其接受和返回的数据类型，包括有关敏感数据的信息。因此，MINT 会自动检查敏感数据不会通过公共端点发送（默认情况下所有端点均为私有），并且私有端点也可以被标记为不敏感（或特定级别）。这使得可以立即对可能公开暴露敏感数据的端点进行静态检查，也使得可以分析任何工作流，以确定其是否能够访问敏感数据；如果可以，则确定是哪些数据以及潜在风险是什么。例如，当智能体访问包含敏感数据的端点时，运行时可以自动隔离该智能体的任何输出，因为这些输出现在可能已被敏感细节污染。

除了监控数据流和敏感性外，MINT 运行时还会根据 URI 自动对执行进行沙箱隔离

其声明将访问的资源。在图 4 中，转账任务声明仅访问收款人的账户资源。因此，该实现无法恶意访问余额，也无法从付款人账户转出资金。这项技术还可应用于任何可映射到 URI 的资源，例如使用 glob语法将特定的一组 REST资源列入白名单，以防止数据外泄到未授权端点；或者禁止访问 `file:///tmp/app_name/`文件夹之外的路径，从而允许写入/读取中间文件，同时确保不会访问用户数据。这确保即使智能体遭到入侵或存在恶意行为，也无法访问其声明范围之外的资源。

虽然沙箱隔离的概念广为人知，但这里的关键洞见在于使用 URI 和 Glob 来规约资源并实现沙箱隔离。与过去使用自定义资源类型和访问语言的做法不同，后者历来难以标准化、理解和使用，URI 和 Glob 能够自然地匹配 RESTful 系统所采用的资源模型。因此，无需针对每种类型（文件、REST端点、数据库、等等）定制插件或扩展，就能以简单、统一且强大的方式，为各种资源类型指定并强制执行资源访问策略。

最后，MINT 运行时还为智能体工作负载提供动态监控和强制执行系统。由于 BOSQUE 和 BAPI 规范能够高效地执行前置/后置条件、不变量和断言，MINT 系统可以在运行时对它们进行监控和强制执行。任何此类条件失效都会触发对违规任务的安全中止；此外，还可以配置为触发其他缓解措施，例如回滚状态更改、记录相关数据，甚至生成离线的时间旅行可调试转储 [4]。这为防止智能体行为不当提供了强有力的防线。这种多层且明确的安全性与防护方法意味着，即使恶意行为者能够利用并绕过某一层防护，仍会受到其他层的限制：这些防护要么阻止攻击，要么迫使攻击者为串联利用链中的攻击寻找多个绕过方法。

6 讨论

一个关键问题是，未来足够强大的 AGI智能体是否还需要或能否受益于智能体注入式软件生态系统的特性，抑或只要经过充分训练并具备足够能力，它就能在任何语言或平台上运行?

设想一种极限情形：假定未来的 AGI智能体能力达到专家级人类开发者的水平，即跻身全球前 10%，那么我们可以通过提出一个类似的问题来回答：这位专家能否从本文概述的特性中受益？答案是肯定的。事实上，对于我们概述的许多特性，其描述和设计动机之一，正是说明它们如何消除或帮助解决现实开发者当前面临的问题。

当追求三个（或更多）9 的可靠性时，即使很小的故障概率（1% 或更低）也不可接受，因为在这一水平上，几乎任何错误都会超出故障预算。我们会接受智能体有 1% 的概率停止偿还我们的抵押贷款、删除客户数据库，或将医疗信息发布到公共论坛吗？因此，在努力构建高度可靠且值得信赖的智能体时，我们需要设置多层相互重叠的保障措施来提升可靠性，因此放弃任何可能的优势都不明智。

反过来要问的是，本文概述的设计与原则，是否针对大语言模型和基于注意力的模型当前的技术水平过度拟合。当这些模型发生变化或被新的架构取代时，这一系统的设计是否会过时，甚至成为障碍?

与上一个问题一样，首先仍需考虑用专家级人类开发者替代大语言模型这一假设极限。再次可以看到，此处概述的原则并非专为基于注意力的模型或大语言模型设计，而是同样适用于人类以及早期（较简单的）程序合成系统（例如 NLyze [19]）。

此外，系统设计允许透明地替换智能体。智能体调用等功能经过专门设计，避免将参数限定为纯文本；SUNDEW 验证器则可处理任何代码，无论其由谁或什么生成。实际上，智能体可以是参与其中的人类，此时所提出的 AISE 仍然有效且连贯。因此，即使底层技术不断演进，所提出的设计依然可行且有效。

7 相关工作

智能体编程语言： 尽管编程语言设计对 AI 智能体的能力有着重要影响，但专门针对智能体软件开发的编程语言设计研究相对较少。该领域最重要的工作来自 Marron [27, 28] 和 Meijer [30–32]，他们都提出了设计方案，重点探讨编程语言设计如何影响 AI 编程和智能体系统的能力。据我们所知，Universalis 语言（Meijer）尚未公开发布，也未经过正式描述。BOSQUE 语言以及本文提出的扩展（参见数据可用性声明）均为完全开源 [6]，所有开发过程也都公开可查。本文提出的扩展远超这两个系统各自的最先进水平，将设计从核心编程语言或工具扩展为一个涵盖部署与编排的完整生态系统。

可验证人工智能编程：可验证人工智能程序合成一直是形式化方法领域研究的重点，研究人员投入了大量工作，使用 Lean [23]和 Dafny [10]等语言，对 AI生成代码进行形式化规约并验证其正确性。这些系统在验证 AI生成代码的正确性方面展现出良好前景，但也凸显了现有工具的局限，尤其是所需规格说明的复杂度和规模。实践中，经验证的人工智能程序合成目前最先进的成果仅适用于小型、独立的程序和 LeetCode 风格问题 [25, 33, 46, 48, 49],；即便如此，规格说明往往与代码本身一样庞大（甚至更大），也一样复杂 [12]。这些挑战凸显了 AISE 愿景所需的核心组成部分：精简的（侧重部分正确性）规约语言、多模态支持，以及可扩展的机械化验证工具。

规范与协作：规范和需求收集是实现软件开发中人类与人工智能智能体有效协作的一项重大挑战

在软件开发中，尤其值得关注的是关于多模态规范以及与以下对象交互的既有研究：终端用户编程 [18, 41] 系统，例如 FlashFill [18] 和 NLyze [19], 以及 Prorogued Programming [1],这些研究展现了让用户以多种形式提供规格说明的潜力。Gulwani 等人的相关研究探讨了将示例和演示用作 AI 智能体规范的形式 [29], 这是降低表达意图（规格说明）难度、并以协作方式与 AI 智能体反复迭代的一条有希望的途径。AISE 愿景在这些既有研究的基础上，将多模态规范与验证深度集成到语言中，并提供一个平台，以支持软件开发中人机协作的新型用户体验和交互范式。

智能体系统：近期关于长时程智能体系统和工具使用的研究，例如 AppWorld [47], LOOP [8] 和 ToolFormer [42],，展示了 AI 智能体在较长时间内执行复杂任务的潜力。然而，这些系统也凸显了上下文管理和工具发现方面的挑战，以及对稳健故障处理与恢复机制的需求。随着任务变得更加复杂，未经增强的系统在较简单任务上的成功率为 70%，但对于涉及更复杂 API（工具）使用的复杂任务（时程更长），成功率会迅速降至 45% 左右。这项工作提出了一系列互补贡献——包括语言设计、机械化验证和运行时特性——旨在与原始智能体能力的提升相结合，打造更稳健、更有效的智能体软件开发生态系统。

8 Onward!

本文概述了构建融入智能体的软件生态系统的愿景和核心设计理念，并介绍了该系统的关键组成部分，从编程语言、机械化验证工具到运行时环境。对于如何创建有效且可信赖的智能体软件系统，我们采取整体性方法，全面考量整个软件技术栈，并协同设计语言、工具和运行时环境，使其协同运作。正如本文所示，这种方法对于应对上下文管理、API/工具发现、规范与需求收集、代码/行动生成以及安全性等方面的挑战至关重要，而这些挑战关系到智能体软件系统能否成功。通过推动这些领域的技术发展，并解决当前智能体部署的关键局限，我们能够打造一个大幅提升智能体软件系统可行性、推动其广泛采用的软件生态系统。

数据可用性

本文所述全部系统的实验版本均可通过主 Bosque GitHub 仓库 https: //github.com/BosqueLanguage公开获取。这些系统正在被积极集成并转换为生产就绪系统，从而将智能体注入式软件生态系统带给各地的开发者（及用户）。

参考文献

[1] Mehrdad Afshari、Earl T. Barr 和 Zhendong Su。2012 年。《通过 Prorogued Programming 解放程序员》。发表于 Onward! [2] Paschal C. Amusuo、Kyle A. Robinson、Tanmay Singla、Huiyun Peng、Aravind Machiry、Santiago Torres-Arias、Laurent Simon 和 James C. Davis。2025 年。《ZT-DJAVA：通过零信任依赖缓解软件供应链漏洞》（IEEE/ACM 第 47 届软件工程国际会议论文集）。[3] Anthropic。2024 年。《模型上下文协议》。https://www.anthropic.com/news/model-context-protocol。[4] Earl T. Barr、Mark Marron、Ed Maurer、Dan Moseley 和 Gaurav Seth。2016 年。《JavaScript/Node.js 的时间旅行调试》（FSE）。[5] Andrew Blinn、Xiang Li、June Hyung Kim 和 Cyrus Omar。2024 年。《使用类型化空洞对大语言模型进行静态上下文化》（OOPSLA）。[6] Bosque Source 2024。《Bosque 编程语言》。https://github.com/BosqueLanguage/。[7] Frederick P. Brooks, Jr. 1987 年。《没有银弹：软件工程的本质与偶然性》。Computer 20（1987）。[8] Kevin Chen、Marco Cusumano-Towner、Brody Huval、Aleksei Petrenko、Jackson Hamburger、Vladlen Koltun 和 Philipp Krähenbühl。2025 年。《面向长时域交互式大语言模型智能体的强化学习》。arXiv:2502.01600 [cs.LG] https://arxiv.org/abs/2502.01600。Clark Barrett、Pascal Fontaine 和 Cesare Tinelli。[9] 2025 年。《SMT-LIB 标准：2.7 版》。https://smt-lib.org/papers/smt-lib-reference-v2.7-r2025-07-07.pdf。[10] Dafny Source 2024。《Dafny 编程语言》。https://dafny.org/。[11] James C. Davis、Christy A. Coghlan、Francisco Servant 和 Dongyoon Lee。2018 年。《正则表达式拒绝服务（ReDoS）在实践中的影响：生态系统规模的实证研究》（ESEC/FSE 2018）。[12] Dodds, Mike。2025 年。《形式化方法的销售：哪些有效，哪些无效》。https://www.galois.com/articles/what-works-and-doesnt-selling-formal-methods。[13] Express.js 2019。《文档》。https://expressjs.com/。[14] Jiazhan Feng、Shijue Huang、Xingwei Qu、Ge Zhang、Yujia Qin、Baoquan Zhong、Chengquan Jiang、Jinxin Chi 和 Wanjun Zhong。2025 年。《ReTool：面向大语言模型战略性工具使用的强化学习》。arXiv:2504.11536 [cs.CL] https://arxiv.org/abs/2504.11536。Roy [15] Thomas Fielding。2000 年。《网络软件架构的架构风格与设计》。博士学位论文。[16] FreeBSD 基金会。2024 年。《FreeBSD Jail 简介》。https://freebsdfoundation.org/freebsd-project/resources/introduction-to-freebsd-jails/。[17] Stephen Goldbaum、Attila Mihaly、Tosha Ellison、Earl T. Barr 和 Mark Marron。2022 年。《面向金融监管和业务平台的高保障软件》（VMCAI）。[18] Sumit Gulwani。2011 年。《使用输入—输出示例自动化电子表格中的字符串处理》。发表于 POPL。[19] Sumit Gulwani 和 Mark Marron。2014 年。《NLyze：通过自然语言对电子表格数据进行交互式编程、分析与操作》。发表于 SIGMOD。[20] Sumit Gulwani、Oleksandr Polozov 和 Rishabh Singh。2017 年。《程序综合》。Foundations and Trends Programming Languages（2017）。[21] Rafael-Michael Karampatsis 和 Charles Sutton。2020 年。《单语句漏洞多久发生一次？ManySStuBs4J 数据集》（MSR）。[22] Andrew Kennedy。2009 年。《计量单位的类型：理论与实践》（CEFP）。[23] Lean Source 2024。《Lean 编程语言》。https://lean-lang.org/。[24] Leonardo de Moura 和 Nikolaj Bjørner。2024 年。《Z3 SMT 定理证明器》。https://github.com/Z3Prover/z3。[25] Chloe Loughridge、Qinyi Sun、Seth Ahrenbach、Federico Cassano、Chuyue Sun、Ying Sheng、Anish Mudide、Md Rakib Hossain Misu、Nada Amin 和 Max Tegmark。2024 年。《DafnyBench：形式化软件验证基准》。arXiv:2406.08467 [cs.SE] https://arxiv.org/abs/2406.08467。[26] Zohar Manna 和 Richard Waldinger。1980。《程序综合的演绎方法》。ACM 编程语言系统汇刊（1980）。[27] Mark Marron。2023 年。《迈向用于推理的编程语言：人类、符号系统与 AI 智能体》（Onward!）。[28] Mark Marron。2024 年。《释放终端用户可编程 AI 的力量》（Onward!）。[29] Mikaël Mayer、Gustavo Soares、Maxim Grechkin、Vu Le、Mark Marron、Oleksandr Polozov、Rishabh Singh、Benjamin Zorn 和 Sumit Gulwani。2015 年。《示例编程中的用户交互式消歧模型》。发表于 UIST。[30] Erik Meijer。2025 年。《从函数挫败到框架灵活性》。Commun. ACM（2025）。[31] Erik Meijer。2025 年。《释放终端用户可编程 AI 的力量》。Commun. ACM（2025）。[32] Erik Meijer。2026 年。《智能体的守护者》。Commun. ACM（2026）。[33] Md Rakib Hossain Misu、Cristina V. Lopes、Iris Ma 和 James Noble。2024 年。《迈向 AI 辅助的已验证 Dafny 方法综合》。ACM on Software Engineering（2024）。

[34] MITRE。2024 年。《CWE-25：最危险的25种软件弱点》。https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html 访问日期：2024-04-08。[35] Morphir 2021。Morphir。https://github.com/finos/morphir。[36] Niels Mündler、Jingxuan He、Hao Wang、Koushik Sen、Dawn Song 和 Martin Vechev。2025 年。《使用语言模型进行类型约束代码生成》（PLDI）。[37] NPM 2023。npm.js。https://www.npmjs.com/。[38] OpenAI。2024 年。Skills。https://developers.openai.com/api/docs/guides/tools-skills/。[39] Oracle。2024 年。《Java 平台：安全管理器》。https://docs.oracle.com/javase/tutorial/essential/environment/security.html。[40] Andrew Rice、Edward Aftandilian、Ciera Jaspan、Emily Johnston、Michael Pradel 和 Yulissa Arroyo-Paredes。2017 年。《检测参数选择缺陷》。ACM 编程语言论文集（2017）。[41] C. Rich 和 R.C. Waters。1988 年。《自动编程：神话与前景》。Computer（1988）。[42] Timo Schick、Jane Dwivedi-Yu、Roberto Dessì、Roberta Raileanu、Maria Lomeli、Luke Zettlemoyer、Nicola Cancedda 和 Thomas Scialom。2023 年。《Toolformer：语言模型能够自学使用工具》。arXiv:2302.04761 [cs.CL] https://arxiv.org/abs/2302.04761

[43] Torsten Scholak、Nathan Schucher 和 Dzmitry Bahdanau。 [无日期。]。PICARD：用于从语言模型进行约束自回归解码的增量解析。发表于*EMNLP*。[44] Joykirat Singh、Raghav Magazine、Yash Pandya 和 Akshay Nambi。2025。通过强化学习实现大语言模型的智能体推理与工具集成。arXiv: 2505 01441 [ ] 2505 01441 。[45] cs.AIhttps://arxiv.org/abs/ 。Armando Solar-Lezama、Gilad Arnold、Liviu Tancau、Rastislav Bodik、Vijay Saraswat 和 Sanjit Seshia。2007。草图模板*(PLDI)*。[46] Chuyue Sun、Ying Sheng、Oded Padon 和 Clark Barrett。2024。Clover：闭环可验证代码生成*(SAIV)*。[47] Harsh Trivedi、Tushar Khot、Mareike Hartmann、Ruskin Manku、Vinty Dong、Edward Li、Shashank Gupta、Ashish Sabharwal 和 Niranjan Balasubramanian。2024。AppWorld：用于交互式编程智能体基准测试的可控应用与人类世界*(ACL)*。[48] Haoxin Tu、Huan Zhao、Yahui Song、Mehtab Zafar、Ruijie Meng 和 Abhik Roychoudhury。2025。智能体程序验证。arXiv：2511.17330 [cs.SE] https://arxiv.org/abs/2511.17330[49] Zhe Ye、Zhengxu Yan、Jingxuan He、Timothe Kasriel、Kaiyu Yang 和 Dawn Song。2025。VERINA：可验证代码生成基准测试。arXiv：2505.23135 [cs.LG] https://arxiv.org/abs/2505.23135