从语法增量到令牌掩码：运行时语法演化下的增量约束解码

Hantao Hua[1], JimingSu[2], Yonggang Che[1] & Feng Zhu[2][1]国防科技大学计算机科学与技术学院[2]国防科技大学系统工程学院{ht hua,zhufeng}@nudt.edu.cn

arXiv:2611.32807v1 [cs.CL] 24 Nov 2026

摘要

语法约束解码使大语言模型能够生成满足形式约束的输出，但在代码和领域专用语言生成中，相关约束并不总是在解码前就完全可用，而可能随着新的声明、推断的类型信息或运行时分析反馈逐步出现。现有结构化解码引擎能够高效处理固定的、预先声明的或此前见过的约束结构；然而，当首次出现的语法更新改变了由已提交前缀所确定的解析器状态时，直接的解决方案是基于更新后的文法重构约束状态，从而重复执行此前已针对该前缀完成的解析工作。我们提出**GramDelta**，一个支持仅追加式运行时语法更新的约束解码增量框架。GramDelta 在已提交前缀上维护解析状态，并仅沿历史依赖关系传播由新增语法规则引起的变更，从而避免前缀重放。随后，它通过对共享历史图表进行无需编译的因子化投影，将更新后的解析状态映射为精确的下一令牌掩码，避免针对每次更新重新编译掩码缓存。因此，GramDelta 在保持完整重构结果的同时，使更新成本取决于语法变更的实际影响。在四种模型配置上的实验表明，在没有更新时，GramDelta 带来的额外开销可以忽略不计；在持续发生运行时语法演化的情况下，相较于最强的重构基线，其首次出现语法更新延迟的中位数降低了 10.7–11.5×，相较于无约束解码的额外每输出令牌时间开销降低了**53.1–81.9%**。这些结果使约束能够随着生成前缀逐步出现和演化时，仍可实现高效的约束解码。

1 引言

语法约束解码（Geng 等人，2023；Scholak 等人，2021；Dong 等人，2025；Hua 等人，2026）使大语言模型（LLM）能够可靠地生成满足形式约束的结构化输出，其方法是在解码过程中限制无效令牌。然而，在代码和领域特定语言生成中，有效约束并不总是在解码开始前就完全确定，而可能随着生成的进行逐步显现（Li et al., 2026）。新生成的声明可以引入此前不存在的标识符和作用域（Zhang et al., 2026）；类型推断可以进一步限制可接受表达式（M¨undler 等人，2025）；静态或运行时分析（Li 等人，2025）则可以确定哪些字段、应用程序接口或程序操作对于当前部分程序是有效的。这些场景的共同特点是，决定有效延续的信息只有在生成上下文建立后才会变得可用。

近期研究从多个方向支持此类运行时依赖约束。结构化生成引擎可以在动态选择的语法组件之间进行分派，并复用相关约束结构之间的计算（Li et al., 2026）；与此同时，代码生成方法可以从不断演化的程序环境中实例化或推导约束（M¨undler et al., 2025；Zhang et al., 2026）。这些系统表明，约束解码不必作用于一种在生成前其有效结构就已完全固定的语言。

然而，现有方法主要关注如何选择、构造或应用运行时约束，而不是如何更新已针对某个已提交前缀计算出的解析器状态。当首次出现的语法增量被引入时，新添加的产生式可能会改变这一未改变前缀的解析结果。因此，完整重构会重新处理历史输入，其中包括不受此次更新影响的计算。

为避免这种冗余工作，我们提出 **GramDelta**，一种适用于加法式运行时语法更新的增量式受约束解码框架。GramDelta 保留已提交前缀对应的 Earley 状态。当新的产生式到达时，它会在能够激活该产生式的历史等待项处将其设为种子，仅传播新近启用的推导，并通过无需编译的因子化投影，将更新后的状态映射为精确的下一令牌掩码：宽泛的词法延续通过与语法无关的词汇位集合解析，而跨边界多字节令牌则在共享的历史图表上进行验证，无需重新编译语法绑定缓存。

GramDelta 通过三个组成部分解决这一问题：

- 我们刻画了加法式语法更新的历史入口，并证明最早入口之前的图表保持不变，因此可以直接为更新设置种子，而无需重放已生成的前缀。
- 我们为 Earley 预测、扫描和完成推导出半朴素增量规则，仅传播涉及新事实的计算，并证明其不动点与更新后语法下的完整重构结果完全匹配。
- 我们设计了一种无需编译的因子化令牌投影器，将与语法无关的词法补集位集与起始位置一致的字典树遍历结合起来，并在历史图表间共享，从而在无需为每次更新重新编译缓存的情况下，精确保留多符号令牌的有效性。

我们在多个模型系列、语法结构和运行时更新工作负载上评估了 GramDelta。在所有评估案例中，GramDelta 均保持了与完整重构等价的解析器状态和令牌有效性；在没有更新时仅增加可忽略的开销，同时相较于最强的重建基线，将首次出现语法更新延迟的中位数降低了 10.7–11.5×，并将持续运行时语法演化带来的额外端到端每输出令牌时间开销降低了 **53.1–81.9%**。

2 背景与相关工作

2.1 语法约束解码

语法约束解码会在每个生成步骤限制候选令牌，使生成的输出始终符合某种形式语言（Willard & Louf, 2023; Koo et al., 2024）。给定一个字节字母表上的语法 G Σ、由当前生成前缀 $x_{<t} \in \Sigma^t$ 诱导的字节序列，以及分词器词汇表 $\mathcal{V} = \mathcal{V}_{\text{byte}} \sqcup \mathcal{V}_{\text{stop}}$ （不包括空载荷特殊令牌），令 $b(v) \in \Sigma^+$ 表示令牌 $v \in \mathcal{V}_{\text{byte}}$的字节序列。令牌掩码$M_t \in \{0,1\}^{|\mathcal{V}|}$ 定义为

$$M_t[v] = 1 \iff \begin{cases} \exists y \in \Sigma^*, \quad x_{<t} b(v) y \in L(G), & v \in \mathcal{V}_{\text{byte}}, \\ x_{<t} \in L(G), & v \in \mathcal{V}_{\text{stop}}. \end{cases}$$

因此，运行时的核心任务是根据当前前缀确定有效的下一个令牌。现有系统通过组合使用自动机或解析器状态、分词器感知预处理、字典树遍历、缓存和语法编译等技术来加速这一操作。例如，基于自动机的方法会预先计算分词器与语言之间的兼容性；XGrammar 会编译语法结构，以便在运行时高效生成掩码（Dong 等人，2025年；Li et al., 2026）；Pre^3 (Chen et al., 2025) 利用 LR(1) 语法的确定性下推转换；而基于 Earley 的系统（如 Formatron（Sun 等人，2025年）和 llguidance（Guidance AI，2025年））则通过增量推进解析器状态来构建令牌有效性。当相关语法固定或已可用时，这些方法能够显著降低约束解码的开销。

相反，我们研究的是：当新的语法结构只有在当前前缀已按照早期语法处理完毕后才可用时，会发生什么。

2.2 运行时语法演化与状态维护

近期的受约束解码系统支持多个层面的动态结构。llguidance 将 Earley 识别器与基于导数的词法分析器及分词器字典树遍历结合起来，为给定语法计算令牌掩码（Guidance AI, 2025）。XGrammar-2 面向动态智能体工作负载：TagDispatch 在生成过程中切换语法组件，同时通过跨语法缓存、自适应 Earley 词元掩码缓存和即时编译，在相关结构之间复用计算结果（Li et al., 2026）。解码时语法则通过从不断演变的运行时环境 Γ 中实例化语法片段，提供了另一种动态形式；这使得应用于后续区域的约束能够取决于已经生成的声明（Zhang et al., 2026）。其他代码生成方法也会根据类型、作用域或程序状态推导上下文相关约束（M¨undler et al., 2025; Li et al., 2025）。总体而言，这些方法提高了生成时约束的构建、选择或执行效率。

当语法增量 ΔG 在前缀 $x_{<t}$ 已依据当前语法 G处理完毕后才被引入时，变化便会出现在另一个维度。这种更新不仅会改变后续需要执行的约束，还可能改变由 同一前缀在 $G \cup \Delta G$下所诱导出的解析状态。现有的复用机制无法直接跨语法版本维护这一历史状态：XGrammar-2 会复用语法子结构和词元掩码缓存，llguidance 会在给定语法下基于已提交词元推进其识别器，Pre^3 会针对固定的 LR(1) 语法预计算下推转换，而解码时语法则会为后续生成区域实例化相应片段。因此，在这种延迟语法更新下，要获得等价于依据更新后的文法处理同一前缀所得到的状态，仍需要重新建立其解析计算，重复已经完成的工作。

解析研究提供了几种密切相关的增量复用形式。增量解析会在输入发生变化时复用解析结果（Wagner & Graham, 1998）；反射式语法会在解析过程中引入具有作用域的语法扩展（Stansifer & Wand, 2011）；而 DEXTER 会在添加或移除产生式时增量更新 LR goto 图（Cazzola & Vacchi, 2014）。这些方法会增量维护依赖于输入的解析结果或依赖于语法的解析器结构，但不会维护这样一种依赖于前缀的历史状态：当之后添加产生式时，该状态必须被修订，而输入本身保持不变。因此，它们无法直接消除上述延迟更新场景所引发的历史重计算。

3 GramDelta

给定当前语法 G、已提交前缀 $x_{<t}$，以及已经为该前缀具体化的解析状态，运行时可能会获得加法式语法更新 ΔG。我们的目标是在不改变已提交前缀的情况下，获取更新后的文法所诱导的解析器状态和下一令牌掩码

$$G' = G \cup \Delta G$$

而不更改已提交前缀。

图1对比了达到这一相同目标的两种方式。参考重建路径在 G' 下构建更新后的识别器，从头开始重放已提交前缀，然后将重建的解析器状态投影为下一令牌掩码。GramDelta 则复用已经具体化的历史状态，并分三个阶段对其进行更新。

首先，历史增量定位查询持久化的历史等待依赖，以确定新添加的产生式可以进入旧 Earley 闭包的位置。最早的此类位置 j_Δ 给出了历史变化的左边界，而所有匹配的历史等待项共同构成种子集 S_Δ。其次，增量 *Earley* 闭包仅通过事件驱动工作列表传播新启用的预测、扫描和完成推导，从而生成更新后的 Earley 状态，而无需重放前缀。第三，因式分解令牌投影将与语法无关的词法补集位集合，与基于共享历史图表进行的起点一致的字典树遍历相结合，在无需重新编译语法绑定缓存的情况下，生成精确的下一令牌掩码。

3.1 运行时更新模型

令 G 表示当前的上下文无关文法，令 $x_{<t}$ 表示由已提交生成前缀诱导的字节序列。为确保活动项目能够进行有限的终结符补全，演绎操作

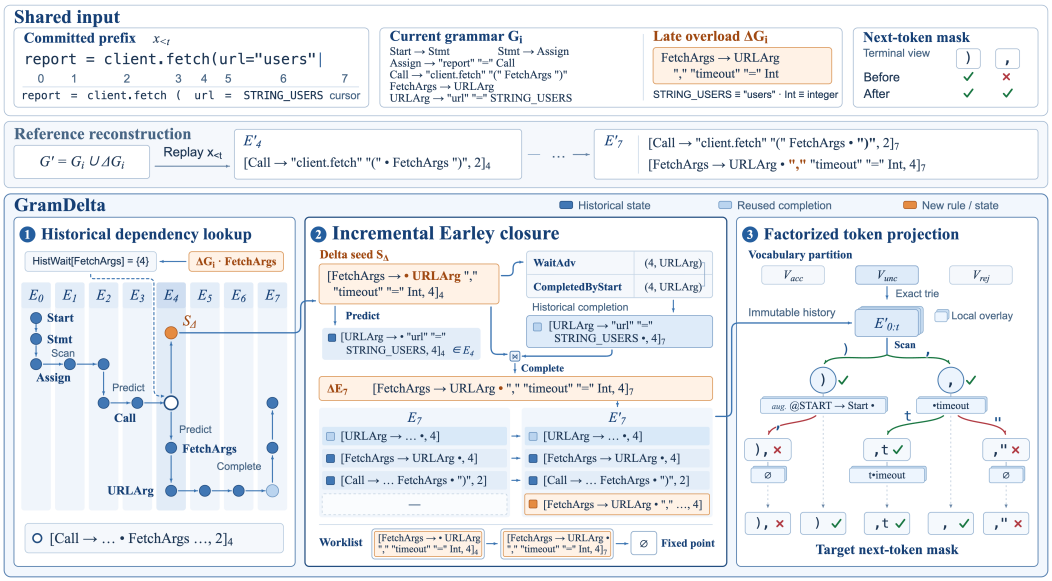


图1：GRAMDELTA 概览。顶部面板展示了已提交代码前缀、当前文法、一次延迟的文法更新，以及由此导致的有效下一令牌集合变化。中部面板描绘了参考重建路径，该路径在更新后的文法下重放已提交前缀。底部面板展示了增量式 GRAMDELTA 流水线：历史依赖查找找到最早受影响的等待位置，增量 Earley 闭包仅传播新启用的状态，同时复用历史完成项；因式分解令牌投影则计算精确的更新后的下一令牌掩码。已完成的增广项 @START → Start • 表明，在消耗右括号后，该语句已被完全接受。

在可生成子语法 $G_{\text{prod}} \subseteq G$上，其右侧非终结符可以推导出终结符串（通过可生成符号注册表单调跟踪；见附录 A.1）。GramDelta 保留历史 Earley 状态

$$E_{0:t} = (E_0, \ldots, E_t)$$

其中一个项目

$$[A \to \alpha \bullet \beta, o]_k \in E_k$$

表明产生式 $A \to \alpha\beta$ 在位置 o处开始，并且 α 已匹配 $x_{o:k}$。我们使用标准的预测、扫描和完成推导（Earley，1970年）。

此前未见的增量语法更新 ΔG 可能只有在 $x_{<t}$ 提交后才可用，例如新声明的标识符、推断出的成员签名或已发现模式进入作用域时。我们考虑产生式添加，而不删除或修改现有产生式

$$G' = G \cup \Delta G \qquad x_{<t} \text{ remains fixed}$$

更新后目标为

$$E'_{0:t} = \text{Earley}(G'_{\text{prod}}, x_{<t})$$

以及由该更新状态 M'_t 所诱导的下一令牌掩码。

由于可生成成员关系和 Earley 演绎在增加产生式的情况下具有单调性

$$E_k \subseteq E'_k \qquad \Delta E_k = E'_k \setminus E_k$$

GramDelta 将运行时语法演化视为对已物化不动点的增量维护：它只计算获得 $E'_{0:t}$所需的新事实 $\Delta E_{0:t}$ ，同时保留已提交前缀和现有历史状态。

3.2 历史增量定位

当更新到达时，旧的 Earley 闭包已经完成物化。GramDelta 首先确定新激活的产生式可以从何处进入这次历史计算。

令 $\Delta P_{\text{act}} = P_{\text{prod}}(G') \setminus P_{\text{prod}}(G)$ 表示在 $G' = G \cup \Delta G$ 下新近变得可生成的所有产生式，包括 ΔG 中的可生成规则，以及 G 中此前处于休眠状态、其右部通过 ΔG 变得可生成的任何规则。对于非终结符 A，定义其历史等待位置

$$\mathcal{W}_E(A) = \{k \mid \exists [B \to \beta \bullet A\eta, o]_k \in E_k\}$$

对于每个新近激活的产生式 $p\colon\ A \to \gamma \in \Delta P_{\text{act}}$，相应的增量种子为

$$\text{Seed}(p, E) = \{[A \to \bullet\gamma, k]_k \mid k \in \mathcal{W}_E(A),\ [A \to \bullet\gamma, k]_k \notin E_k\}$$

and

$$S_\Delta = \bigcup_{p \in \Delta P_{\text{act}}} \text{Seed}(p, E)$$

S_Δ 是精确的，而非启发式的：它刻画了新近激活的产生式首次离开旧 Earley 闭包时所经过的入口点。

命题 1（首个新事实刻画）。每一个

$$e \in E'_{0:t} \setminus E_{0:t}$$

存在一种 Earley推导，其位于旧闭包之外的第一个事实属于 S_Δ。因此

$$S_\Delta = \varnothing \quad \Longrightarrow \quad E'_{0:t} = E_{0:t}$$

证明以及相应的历史左边界性质见附录A.2。

为避免更新到达时扫描历史图表，GramDelta 在正常解码期间维护一个按非终结符索引的关系 HistWait$[A]$。每当位置 k 包含一个等待 A的项目时，就会在该关系中登记 k。因此，新激活的产生式 $A \to \gamma$ 可以直接从 HistWait$[A]$实例化其种子。

所保留的依赖是语义层面的，而非表示层面的：历史图表行可以组织到共享不可变页面中，前提是任何可能因未来语法更新而重新激活的等待依赖或完成依赖都不会被不可逆地丢弃。

3.3 增量 Earley 闭包

种子集 S_Δ 标识出语法更新首次进入历史计算的位置。随后，GramDelta 将 $E_{0:t}$ 视为一个已物化的 Earley不动点（Pereira & Warren，1983），并仅传播由这些种子新启用的推导（Bancilhon & Ramakrishnan，1986）。

因子化预测。令 U 编码活动非终结符等待

$$U(k, A) = 1 \iff E_k \text{ contains an item waiting for } A$$

并令 V将非终结符映射到产生式入口状态

$$V(A, q_p^0) = 1 \iff q_p^0 \text{ is the entry state of } p : A \to \gamma \in P_{\text{prod}}$$

于是，预测可以写成因子化关系

$$\mathsf{P} = UV$$

语法更新通过 ΔV扩展 V，而增量传播可能引入额外的等待ΔU。因此，新增加的预测工作为

$$\Delta\mathsf{P} = U\Delta V \cup \Delta U V' \qquad V' = V \cup \Delta V$$

第一项正是第 3.2 节中的历史种子关系；第二项则传播增量闭包所创建的等待项。

算法 1 GramDelta 识别器更新

要求： 持久状态 $\mathcal{R}$、已提交前缀 $x_{<t}$、新产生式 ΔG
确保： 更新后的 Earley 状态

$\Delta P_{\text{act}} \leftarrow$添加产生式$(\Delta G)$
对于所有 $p: \quad A \to \gamma \in \Delta P_{\text{act}}$ **执行**
  对于所有 $k \in \texttt{HistWait}[A]$**执行**
    插入$([A \to \bullet\gamma, k]_k)$
  结束循环
结束循环
当WorkListNonEmpty**时执行**
  $e \leftarrow$ 弹出工作列表
  如果 e 正在等待非终结符 B，位置为 k **则**
    将 k 注册到 HistWait$[B]$，并将 e 注册到 WaitAdv$[(k, \ B)]$
    INSERT（所有可生成的预测后继项 B）
    INSERT（所有与按起点完成项 $[(k, \ B)]$连接的项）
  否则，如果 e 正在等待终结符 τ，位于 $k < t$ **则**
    如果 MATCHES$(\tau, \ x_k)$ **则**
      插入（e的扫描后继）
    结束条件判断
  否则，如果 e 从 o完成 B **则**
    将 e 注册到 CompletedByStart$[(o, \ B)]$ 中
    将 e 与 WaitAdv$[(o, \ B)]$ 合并
  结束条件判断
结束循环

稀疏扫描。 令 $\mathsf{S}_a(q, \ q^+)$ 表示从带点状态 q 消耗终结符或词法符号 a 后转移至其后继状态 q^+ 的终结符转移。令 ΔT_k 表示定义在带点状态和起点之上的稀疏关系，其定义为

$$\Delta T_k(q, o) = 1 \iff (q, o, k) \in \Delta E_k \text{ and } q \text{ expects a terminal or lexical transition}$$

由于已提交的输入保持不变，语法更新之前已存在的终结符期待事实均已完成扫描。因此，只有 ΔT_k 需要新的扫描工作

$$\Delta E_{k+1}^{\text{scan}} = \Delta T_k \, \mathsf{S}_{x_k}$$

起点敏感完成。 完成操作将位置 j 处等待 B 的等待项，与一个起始于 j的已完成 B 成分进行连接。GramDelta 为连接的两侧维护 WaitAdv$[(j, B)]$ 和 CompletedByStart$[(j, B)]$。新生成的等待项会查询已有的完成项，而新生成的完成项会查询已有的等待项。因此，无论哪一侧后到，都会触发缺失的演绎，而无需重新扫描历史图表。

增量不动点。 每个候选事实都会经过图表成员去重。尚未见过的事实只会插入一次，并更新相应的稀疏依赖关系，且可能触发进一步的预测、扫描或完成推论。当工作列表变为空时，传播便会终止。

上述三个增量算子在算法 1 的事件驱动过程中组合使用。语法更新首先插入第 3.2 节中确定的精确历史种子。此后，每个新插入的事实仅触发由它提供此前缺失前提的预测、扫描和完成推论。成员去重确保每个新的 Earley 事实在工作列表中最多处理一次。

算法 1 中两个"完成"方向都至关重要：新的等待项必须与先前已物化的完成项相匹配，而新的完成项必须与先前已物化的等待项相匹配。因此，该过程从 S_Δ 开始执行依赖驱动传播，而不是从受影响的位置重新回放前缀。

定理 1（精确增量闭包）。 如果

$$E_{0:t} = \text{Earley}(G_{\text{prod}}, x_{<t})$$

更新前和 $G' = G \cup \Delta G$，那么终止时得到的状态满足

$$\widehat{E}_{0:t} = \mathrm{Earley}(G'_{\mathrm{prod}}, x_{<t}) = E'_{0:t}$$

完整的增量推导、证明和事件驱动过程见附录 A.3 和 B.1。

3.4 因式分解令牌投影

增量 Earley 闭包会生成更新后的历史解析状态 $E'_{0:t}$，GramDelta 据此计算更新后的下一令牌掩码 M'_t。由于一个令牌可能跨越多个语法转换，并触发起始于历史位置 $o \le t$ 的成分完成，因此解析器前沿未发生变化，并不意味着之前的令牌掩码仍然有效。因此，GramDelta 直接根据更新后的历史投影令牌，而不重放已提交前缀。

词汇表级补集划分。 GramDelta 预先计算词汇表级符号包含集合

$$\mathcal{I}[\sigma] = \{v \in \mathcal{V}_{\mathrm{byte}} \mid \sigma \in b(v)\},$$

以及与完整词汇原子对齐的令牌子集 $\mathcal{V}_{\mathrm{closed}}$。根据可扫描终结符集合 $\Sigma_{\mathrm{first}}(E'_t)$以及由活动循环保证的可重复终结符集合 Σ_{loop}，词汇表被划分为直接接受、拒绝和不确定的令牌：

$$\begin{aligned}
\mathcal{V}_{\mathrm{acc}} &= \mathcal{V}_{\mathrm{closed}} \setminus \bigcup_{\sigma \notin \Sigma_{\mathrm{loop}}} \mathcal{I}[\sigma], \\
\mathcal{V}_{\mathrm{rej}} &= \{v \in \mathcal{V}_{\mathrm{byte}} \setminus \mathcal{V}_{\mathrm{acc}} \mid b(v)_0 \notin \Sigma_{\mathrm{first}}(E'_t)\}, \\
\mathcal{V}_{\mathrm{unc}} &= \mathcal{V}_{\mathrm{byte}} \setminus (\mathcal{V}_{\mathrm{acc}} \cup \mathcal{V}_{\mathrm{rej}}).
\end{aligned}$$

当不存在活动循环时，$\mathcal{V}_{\mathrm{acc}} = \varnothing$。此划分仅作为快速筛选机制；所有不确定的候选项均由精确的解析器演绎处理。

共享历史的试探性字典树演绎。 对于 $\mathcal{V}_{\mathrm{unc}}$，GramDelta 在候选令牌字节序列的前缀字典树 $\mathcal{T}$上执行试探性 Earley 演绎。深度为 d的每个节点 u表示扩展 $x_{<t}\mathrm{str}(u)$，并在位置 $t+d$维护一个试探性解析状态。字典树根节点直接复用更新后的历史状态 $E'_{0:t}$。沿边推进 $u \xrightarrow{\sigma} w$ 时，在 G'下执行 Scan、Complete 和 Predict，随后执行完整的 Earley 闭包。关键在于让候选分支共享历史解析基底。所有分支都将 $E'_{0:t}$ 视为不可变：试探性完成可以读取历史等待项，但新推导出的事实仅作为分支局部增补，记录在活动路径所经过的位置上。因此，公共字节前缀只需评估一次，而不同分支的增补仍彼此隔离，从而避免冗余演绎和历史状态污染。

令 E_{u_v} 表示与令牌 v 对应的字典树叶节点处的推测状态。如果该状态包含一个期待后续终结符的项目，或包含已完成的开始产生式项，则候选项将被接受：

$$\Pi(E'_{0:t}, \mathcal{V}_{\mathrm{unc}}) = \{v \in \mathcal{V}_{\mathrm{unc}} \mid E_{u_v} \text{ contains a terminal-expecting item or } [S' \to S\bullet, 0]\}.$$

只有当当前状态已包含已完成的开始产生式项时，停止令牌才有效：

$$M_t^{\mathrm{stop}} = \{v \in \mathcal{V}_{\mathrm{stop}} \mid [S' \to S\bullet, 0] \in E'_t\}.$$

由此得到的令牌掩码为

$$M'_t = \mathcal{V}_{\mathrm{acc}} \cup \Pi(E'_{0:t}, \mathcal{V}_{\mathrm{unc}}) \cup M_t^{\mathrm{stop}}.$$

命题 2（精确因子化 TokenProjection）。对于任意有限上下文无关文法 G' 和已提交前缀 $x_{<t}$，如果 $E'_{0:t}$ 是文法 G' 下完整且正确的历史 Earley 状态，快速接受集合满足其循环保持不变量，并且字典树推导在每个字节后都执行完整的 Earley 闭包，那么得到的令牌掩码与在更新后的文法下通过完整重构得到的令牌掩码相同：

$$M'_t = M_t(G', x_{<t}).$$

证明和持久化数据结构表示见附录 A.4 和 B.2。

4 实验

我们评估 GramDelta 是否能够降低首次出现的运行时语法演化成本，以及这种降低是否能够转化为完整服务请求中更低的开销。所有方法均在相同的 SGLang Zheng 等人（2024年）服务栈中，采用统一的延迟更新协议进行评估。

4.1 实验设置

基线。 我们将 GramDelta 与内部和外部基线进行比较：**(i) Fresh**，在同一原生 Earley 引擎内重建识别器并重放已提交前缀，从而将增量状态维护与实现差异区分开来；**(ii)XGrammar** （Dong 等人，2025年；Li 等人，2026），分别在不使用跨更新编译缓存和使用持久化热编译缓存的情况下进行评估；**(iii)`llguidance`**（Guidance AI，2025年），通过符合惯例的正则表达式词法分析器文法复用持久化的 ParserFactory。在我们的首次出现后期更新协议下，重建基线会实例化更新后的匹配器并重新处理已提交前缀，而 GramDelta 则将语法增量直接应用于其持久化的请求局部识别器状态。

模型与工作负载。 我们评估 Qwen3-8B（Yang 等人，2025年）、Qwen3-32B、Granite-4.1-3B（IBM Research，2026）和 Granite-4.1-8B，涵盖多种模型规模和分词器系列。工作负载包括轻量级演化、受控的局部/递归上下文无关文法更新，以及紧凑的渐进式 BFCL 工具调用轨迹（Patil 等人）（$K = 4$–32 次更新）；还包括持续的大型语法演化（受 M¨undler 等人（2025年）和 Zhang 等人（2026年）启发的 CodeEvo 声明驱动代码合成，包含 $\sim$2.8K 条基础规则和 $K = 4$–32 次更新），并设有静态无更新对照（$K = 0$）。每个语法增量仅在指定的运行时位置变为可见；未来的语法片段绝不预先加载。所有模型均在单个 NVIDIA RTX PRO 6000 Blackwell GPU 上运行，每个请求仅进行一次 KV 缓存预填充，并且语法更新时不重置 KV 缓存（附录 C）。

测量协议。 对于首次见到的语法更新，更新延迟 是从 ΔG_i 对约束后端可见的时刻起，到令牌掩码可用为止的实际时间间隔；其中不包括模型前向执行，但包括状态维护或匹配器构建、所需的前缀重新处理以及令牌投影。

每输出令牌时间 （Time PerOutput Token）根据完整的解码请求进行测量，因此包括模型执行、稳态约束解码以及运行时更新处理。我们报告相对于无约束解码的 TPOT 开销：

$$\text{TPOT Overhead} = \frac{\text{TPOT}_{\text{method}} - \text{TPOT}_{\text{unconstrained}}}{\text{TPOT}_{\text{unconstrained}}} \times 100\%.$$

标题性比较采用共享令牌协议（每个单元进行 20–30 次配对运行），以确保所有后端都遵循完全一致的已提交词元序列和更新时间顺序。

4.2 主要结果

表 1 报告了共享令牌延迟更新协议下四种模型配置的端到端 GPU 解码结果（BFCL、8K 前缀、 $K = 32$ 首次见到的更新； $K = 16$、CodeEvo 和紧凑的 512 令牌轨迹在附录 D 中呈现出一致的行为）。对于每次首次见到的语法更新，所有受约束后端都处理完全相同的已提交前缀和更新序列，而不会重置模型 KV 缓存。

首次见到的更新延迟。在全部四种模型中，GramDelta 在 $K = 32$ 下的中位更新延迟为 3.748–3.817 毫秒，相较于同引擎 Fresh 重构，成本降低了 10.89–11.27×。与重构基线不同，它无需重放已提交前缀，也无需重新编译更新后的文法。

端到端解码成本。GramDelta 相较于无约束解码仅增加 1。45–1。53 毫秒/令牌的开销，在四个模型中对应 2。4–10。0% 的每输出令牌时间开销。它将额外

表 1：四种模型配置下的主要图形处理器解码结果（BFCL、8K 前缀、 K =32 次首次见到的更新）。更新延迟是约束更新时间的中位数；TPOT 开销是相对于同一模型的无约束解码而言的。

模型	方法	更新延迟（毫秒）↓	TPOT（毫秒/令牌）↓	每输出令牌时间开销（%）↓
Qwen3-8B	无约束	–	19.645	0.0
	XGrammar	500.680	124.953	536.1
	llguidance	49.437	27.078	37.8
	Fresh	42.066	28.855	46.9
	GramDelta	**3.817**	**21.174**	**7.8**
Qwen3-32B	无约束	–	61.117	0.0
	XGrammar	526.198	163.645	167.8
	llguidance	51.570	65.603	7.3
	Fresh	41.921	70.624	15.6
	GramDelta	**3.748**	**62.570**	**2.4**
Granite-4.1-3B	无约束	–	15.396	0.0
	XGrammar	550.876	133.650	768.1
	llguidance	49.942	23.884	55.1
	Fresh	41.326	25.458	65.4
	GramDelta	**3.795**	**16.929**	**10.0**
Granite-4.1-8B	无约束	–	20.151	0.0
	XGrammar	562.078	139.269	591.1
	llguidance	53.271	28.138	39.6
	Fresh	42.947	29.955	48.7
	GramDelta	**3.810**	**21.674**	**7.6**

表2：Qwen3-8B 在窄范围本地 CPU 更新、宽范围开放字符串投影和 8K $K = 32$ 图形处理器解码下的组件消融。

变体	窄范围更新（毫秒）↓	宽范围投影（毫秒）↓	图形处理器更新（毫秒）↓	GPUTPOT（毫秒/令牌）↓
Fresh（重建 +回放）	2.398	0.038	41.324	29.659
不含 HistWait 的 GramDelta	0.126	0.036	6.316	21.233
不含共享历史的 GramDelta	13.771	0.752	193.368	242.196
不含补集划分的 GramDelta	0.109	543.081	3.852	21.240
完整 GramDelta	**0.104**	**0.034**	**3.757**	**21.208**

每输出令牌时间开销相较于 llguidance 降低了 67。62–81。93%，后者是该指标上表现最强的重建基线。其他更新次数和轻量级工作负载见附录 D。

4.3 消融研究

表2在 Qwen3-8B 上，使用相同的原生诊断构建，分别针对 CPU微基准测试和 8K $K = 32$ GPU 解码，分析了 GramDelta 的各个算法组件。将同引擎的**Fresh** 重建（窄 CPU 更新耗时 2.398 ms；图形处理器更新耗时 41.324 ms，每输出令牌时间为 29.659 ms/tok）替换为半朴素**增量闭包**，可消除已提交令牌重放，使图形处理器每输出令牌时间降至 21.233 ms/tok。加入**`HistWait` 定位**后，播种 S_Δ 时无需再对历史图表进行线性扫描，使 8K 图形处理器更新延迟进一步从 6.316 ms 降至 3.757 ms（快 $1.68\times$），而四种模型上的静态 $K = 0$ 对照表明，保留 HistWait 和完成索引仅使每输出令牌时间变化 -0.13% 至 $+0.12\%$（附录 D）。

共享历史覆盖层对于高效的推测投影至关重要：禁用后，8K 图形处理器更新延迟将从 3.757 毫秒增加到 193.368 毫秒。补集划分主要用于保护宽词法状态：移除后，宽字符串投影延迟将从 0.034 毫秒增加到 543.081 毫秒，而结构更新成本几乎不受影响。这些组件共同避免了历史重复和不必要的全词汇范围的解析器推导。

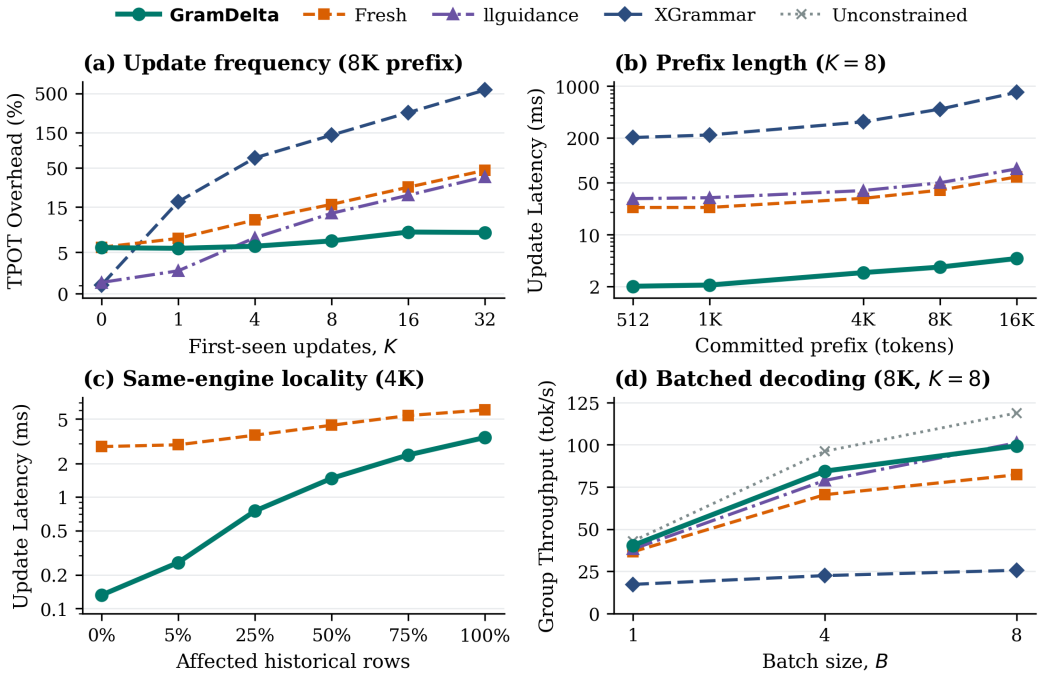


图 2：Qwen3-8B 上的扩展性与局部性表现：(a) 不同首次出现更新计数 $K \in \{0, 1, 4, 8, 16, 32\}$（8K 前缀）下的每输出令牌时间开销；(b) 不同已提交前缀长度（512–16K 令牌，$K = 8$）下的更新延迟；(c) 同引擎更新延迟与受影响的历史图表行数之间的关系（4K 前缀）；以及 (d) 不同批大小 $B \in \{1, 4, 8\}$（8K $K = 8$）下的融合批量解码吞吐量。

4.4 扩展性、局部性与讨论

跨更新频率、前缀长度和批大小的扩展性。图 2 显示，随着运行时更新不断累积，GramDelta 的优势日益明显，而预编译 XGrammar 和 llguidance 在静态或低更新场景下仍保持较低开销。在 $K = 32$，GramDelta 的每输出令牌时间开销为 7.39%，相比之下，llguidance 为 38.33%，Fresh 为 46.76%，XGrammar 为 557.47%。随着前缀从 512 个令牌增长到 16K 个令牌，其更新延迟仅从 2.01 毫秒增加到 4.76 毫秒。在融合批处理下，GramDelta 在较小批大小时领先，并且在 $B = 8$仍与 llguidance 具有竞争力。

对更新局部性的依赖。增量维护的收益取决于更新局部性。随着受影响的历史行从 0% 增加到 100%，GramDelta 的更新延迟从 0。132 毫秒上升至 3。414 毫秒，其相对于 Fresh 的加速比从 $21.4\times$ 缩小至 $1.77\times$。在广泛历史范围的场景中，例如稠密全局 LR(1) 补全或无界右递归，自动机重建可能优于增量 Earley 修复。这影响的是性能而非语义：GramDelta 仍会生成精确状态和令牌掩码。其持久化索引还会带来适度的内存开销（附录 D）。

5 结论

我们提出了 GramDelta，这是一种用于仅追加式运行时语法更新场景下约束解码的增量框架。通过结合历史依赖局部化、半朴素 Earley 闭包和精确因子化令牌投影，GramDelta 无需重放已提交前缀，即可维持与完整重构相同的解析器状态和下一个令牌掩码。其优势取决于更新局部性：范围广泛的变更可能更适合采用重构；而要支持删除和替换，则需要来源感知撤回。

参考文献

Francois Bancilhon 和 Raghu Ramakrishnan。递归查询处理策略入门。载于*1986 年 ACM SIGMOD 数据管理国际会议论文集 - SIGMOD '86*，SIGMOD '86，第 16–52 页。ACM出版社，1986。doi：10.1145/16894.16859。URLhttp://dx.doi.org/10.1145/16894.16859。

Walter Cazzola 和 Edoardo Vacchi。LR GOTO 图的增量扩展与收缩。《信息学学报》，51(7):419–447，2014 年。ISSN 1432-0525。doi：10.1007/s00236-014-0201-2。URL：http://dx.doi.org/10.1007/s00236-014-0201-2。

Junyi Chen、Shihao Bai、Zaijun Wang、Siyu Wu、Chuheng Du、Hailong Yang、Ruihao Gong、Shengzhong Liu、Fan Wu 和 Guihai Chen。Pre^3：支持确定性下推自动机，以实现更快的结构化大语言模型生成。载于 Wanxiang Che、Joyce Nabende、Ekaterina Shutova 和 Mohammad Taher Pilehvar（编），计算语言学协会第 *63* 届年会论文集（第一卷：长篇论文），第 11253–11267 页，奥地利维也纳，2025 年 7 月。计算语言学协会。ISBN 979-8-89176-251-0。doi：10.18653/v1/ 2025.acl-long.551。URL：https://aclanthology.org/2025.acl-long.551/。

Yixin Dong、Charlie F Ruan、Yaxing Cai、Ziyi Xu、Yilong Zhao、Ruihang Lai 和 Tianqi Chen。Xgrammar：面向大语言模型的灵活高效结构化生成引擎。《机器学习与系统会议论文集》，7，2025年。

Jay Earley。一种高效的上下文无关解析算法。《ACM通讯》，13(2):94–102，1970年2月。ISSN1557-7317。doi：10.1145/362007.362035。URL http://dx.doi.org/ 10.1145/362007.362035。

Saibo Geng、Martin Josifoski、Maxime Peyrard 和 Robert West。无需微调即可用于结构化自然语言处理任务的语法约束解码。载于《2023年自然语言处理实证方法会议论文集》，第10932–10952页，2023年。

Guidance AI。《llguidance：超快速结构化输出》。https://github.com/guidance-ai/llguidance，2025年。软件仓库。

Hantao Hua、Jiming Su、Hao Tang、Yiping Yao 和 Feng Zhu。Gram2token：为大语言模型实现运行时 GPU 原生语法约束解码。载于《第四十三届国际机器学习大会》，2026年。URL https://openreview.net/forum?id=h3K23f6tLU。

IBM Research。Granite 4.1 语言模型。https://huggingface.co/blog/ ibm-granite/granit-4-1，2026年。访问日期：2026-04-28。

Terry Koo、Frederick Liu 和 Luheng He。用于语言模型解码的基于自动机的约束。*arXiv* 预印本 *arXiv:2407.08103*，2024年。

Lingxiao Li、Salar Rahili 和 Yiwei Zhao。通过约束解码生成具有正确性保证的代码。*arXiv* 预印本 *arXiv:2508.15866*，2025年。

Linzhang Li、Yixin Dong、Guanjie Wang、Ziyi Xu、Alexander Jiang 和 Tianqi Chen。Xgrammar-2：面向智能体大语言模型的动态高效结构化生成引擎。载于*ACM* 人工智能与智能体系统大会论文集，第 1009–1022 页，2026 年。

Niels M¨undler、Jingxuan He、Hao Wang、Koushik Sen、Dawn Song 和 Martin Vechev。利用语言模型生成受类型约束的代码。载于《*ACM* 编程语言汇刊》，9(PLDI)：601–626，2025年。

Shishir G Patil、Huanzhi Mao、Fanjia Yan、Charlie Cheng-Jie Ji、Vishnu Suresh、Ion Stoica 和 Joseph E Gonzalez。伯克利函数调用排行榜（BFCL）：从工具使用到对大语言模型的智能体评估。载于第四十二届国际机器学习大会。

FCN Pereira 和 DHD Warren。将句法分析视为演绎。ACL 会议论文集。第二十一届年会，1983 年。

Torsten Scholak、Nathan Schucher 和 Dzmitry Bahdanau。PICARD：对语言模型的受约束自回归解码进行增量式句法分析。载于 Marie-Francine Moens、Xu- anjing Huang、Lucia Specia 和 Scott Wen-tau Yih（编），*2021* 年自然语言处理实证方法会议论文集，第 9895–9901 页，线上及多米尼加共和国蓬塔卡纳，2021 年 11 月。计算语言学协会。doi：10.18653/v1/2021.emnlp-main.779。URLhttps://aclanthology.org/2021. emnlp-main.779/。

Paul Stansifer 和 Mitchell Wand。解析反射式语法。载于第十一届语言描述、工具与应用研讨会论文集，LDTA '11，第 1–7 页。ACM，2011 年 3 月。doi：10.1145/1988783.1988793。URL：http://dx.doi.org/10.1145/1988783. 1988793。

Xintong Sun、Chi Wei、Minghao Tian 和 Shiwen Ni。Earley驱动的动态剪枝，用于高效结构化解码。载于 Aarti Singh、Maryam Fazel、Daniel Hsu、Simon Lacoste-Julien、Felix Berkenkamp、Tegan Maharaj、Kiri Wagstaff 和 Jerry Zhu（编），第 *42* 届国际机器学习会议论文集，第 267 卷机器学习研究论文集，第 57773–57789 页。PMLR，2025 年 7 月 13–19 日。URL：https://proceedings.mlr. press/v267/sun25v.html。

Tim A. Wagner 和 Susan L. Graham。高效灵活的增量解析。*ACM*《编程语言与系统汇刊》，20(5):980–1013，1998年。ISSN 1558-4593。doi：10.1145/ 293677.293678。URL http://dx.doi.org/10.1145/293677.293678。

Brandon T. Willard 和 R´emi Louf。面向大语言模型的高效引导式生成，2023年。URL https://arxiv.org/abs/2307.09702。

An Yang、Anfeng Li、Baosong Yang、Beichen Zhang、Binyuan Hui、Bo Zheng、Bowen Yu、ChangGao、ChengenHuang、Chenxu Lv 等。Qwen3 技术报告。*arXiv* 预印本 *arXiv:2505.09388*，2025年。

Shuoming Zhang、Ruiyuan Xu、Haofeng Li、Qiuchu Yu、Yangyu Zhang、Chunwei Xia、Xiaobing Feng、Chenxi Wang、Huimin Cui 和 Jiacheng Zhao。解码时语法：在语法片段细化序上进行受约束的大语言模型生成。*arXiv* 预印本 *arXiv:2607.18357*，2026年。

Lianmin Zheng、Liangsheng Yin、Zhiqiang Xie、Chuyue Sun、Jeff Huang、Cody H Yu、Shiyi Cao、Christos Kozyrakis、Ion Stoica、Joseph E Gonzalez 等。Sglang：结构化语言模型程序的高效执行。神经信息处理系统进展，37：62557–62583，2024年。

A 形式化细节与证明

A.1 EARLEY 演绎系统

我们将运行时约束语法表示为一个上下文无关文法

$$G = (N, \Sigma, P, S)$$

其中， N 是非终结符集合， Σ是终结符字母表， P 是产生式集合， S 是开始符号。给定一个提交后固定的输入前缀

$$x_{<t} = x_0 x_1 \cdots x_{t-1}$$

Earley 识别器在每个输入位置$k \in \{0, \ldots, t\}$维护一个图表 E_k 。

Earley 项写作

$$[A \to \alpha \bullet \beta, o]_k$$

这表示产生式 $A \to \alpha\beta$ 从输入位置 o开始，并且 α 已匹配区间 $x_{o:k}$。我们将 o 称为 起始位置 ，将 k 称为当前图表位置。图表采用集合语义，因此每个项目最多只会实例化一次。

对于固定的语法和输入，Earley 状态是在以下三条标准推理规则下的最小闭包。

预测。 如果 E_k 中的某个项目期待一个非终结符 B，则 B 的每个产生式都会在相同位置引入一个初始项目：

$$\frac{[A \to \alpha \bullet B\beta, o]_k \qquad (B \to \gamma) \in P}{[B \to \bullet\gamma, k]_k}$$

扫描。 如果位置 k 处的某个项目期待终结符 a，且当前输入符号与 a 匹配，则点号推进到下一个图表：

$$\frac{[A \to \alpha \bullet a\beta, o]_k \qquad x_k = a}{[A \to \alpha a \bullet \beta, o]_{k+1}}$$

对于词法终结符， $x_k = a$ 可以替换为相应的词法匹配关系。以下论证仅依赖于扫描从 k 到 $k+1$ 单调推进这一事实，而不依赖于特定的词法表示。

完成。 如果一个成分 B 从位置 j 开始，并在位置 k 完成，那么 E_j 中所有等待 B 的项目都可以推进：

$$\frac{[A \to \alpha \bullet B\beta, o]_j \qquad [B \to \gamma\bullet, j]_k}{[A \to \alpha B \bullet \beta, o]_k}$$

因此，完成操作以 $(j,\ B)$ 为键：已完成成分的起始位置 j 必须与包含其等待项的图表位置一致。这个依赖关系对起始位置敏感，后续将用于构建增量完成索引。

我们将从增广起始项生成的最小不动点记为

$$[S' \to \bullet S, 0]_0$$

as

$$E_{0:t} = \mathrm{Earley}(G, x_{<t})$$

当前缀被接受为完整字符串时

$$[S' \to S\bullet, 0]_t \in E_t$$

然而，对于受约束解码，我们还会在 E_t 中保留允许进一步终结符延续的不完整状态，因为这些状态决定了下一词元的有效性。

为保证每个活跃项目都允许有限的终结符补全，演绎在可生成子语法 $G_{\mathrm{prod}} = (N_{\mathrm{prod}},\ \Sigma,\ P_{\mathrm{prod}},\ S)$ 上进行，其中 $N_{\mathrm{prod}}(G) = \{A \in N \mid \exists w \in \Sigma^*,\quad A \Rightarrow_G^* w\}$ 且 $P_{\mathrm{prod}}(G) = \{A \to \gamma \in P \mid \gamma \in (N_{\mathrm{prod}}(G) \cup \Sigma)^*\}$。我们考虑如下形式的加法式语法更新

$$G' = (N', \Sigma', P \cup \Delta P, S)$$

同时保持 $x_{<t}$ 不变。由于添加产生式不会移除已有推导，因此 $P_{\mathrm{prod}}(G) \subseteq P_{\mathrm{prod}}(G')$ 和 Earley 闭包都是单调的：

$$E_k \subseteq E'_k, \qquad E_{0:t} = \mathrm{Earley}(G_{\mathrm{prod}}, x_{<t}), \qquad E'_{0:t} = \mathrm{Earley}(G'_{\mathrm{prod}}, x_{<t})$$

因此我们可以写作

$$E'_k = E_k \cup \Delta E_k, \qquad \Delta E_k = E'_k \setminus E_k$$

并完全根据新激活产生式 $\Delta P_{\mathrm{act}} = P_{\mathrm{prod}}(G') \setminus P_{\mathrm{prod}}(G)$ 所引入的新 Earley 事实来刻画此次更新。

A.2 历史增量定位的证明

本节给出第 3.2 节中省略的证明。我们沿用该节定义的符号。

首个新事实刻画的证明。考虑任意有限的 Earley推导

$$f \in E' \setminus E$$

并选择一个推导深度最小的新事实 $g \notin E$。根据最小性，g 的每个 Earley 前提都已属于 E。

如果 g 由 扫描 产生，则其唯一前提属于 E。语法更新不会修改已提交前缀，并且对于这个固定输入，E 已经在扫描下闭合。因此，同一个演绎必然已经在旧闭包中产生了 g，这与 $g \notin E$矛盾。

如果 g 由 完成 产生，则其等待项前提和已完成前提都属于 E。由于旧 Earley 状态在完成下是闭合的，它们的结论也必然属于 E，从而得到同样的矛盾。

因此， g 必然由预测产生。令

$$g = [A \to \bullet\gamma, k]_k$$

由一个等待项推导得到

$$[B \to \beta \bullet A\eta, o]_k \in E_k$$

如果 $A \to \gamma \in P_{\text{prod}}(G)$，则相同的预测推导在旧的可生成语法中已经可用，因此 $g \in E$。所以

$$A \to \gamma \in \Delta P_{\text{act}}$$

由于等待项出现在 E_k中，我们有 $k \in \mathcal{W}_E(A)$，因此 $g \in S_\Delta$。所以，任何产生新 Earley 事实的推导，都必须首先通过第 3.2 节中定义的某个种子离开旧闭包。

历史左边界。 假设 $S_\Delta \neq \varnothing$，并定义最早的种子化图表位置

$$j_\Delta = \min\{k \mid [A \to \bullet\gamma, k]_k \in S_\Delta \text{ for some } A \to \gamma \in \Delta P_{\text{act}}\}$$

推论1（历史左边界）。 j_Δ 之前的任何图表位置都不会因加性更新而改变

$$\forall k < j_\Delta \qquad E'_k = E_k$$

证明。 每个新推导都始于 S_Δ中的一个种子。预测保持当前图表位置不变，扫描前进到下一个位置，而完成则在已完成成分的结束位置产生其结果。三种 Earley 演绎均无法将新推导出的事实传播到早于新推导进入旧闭包的位置。由于 j_Δ 是最早的此类进入位置，因此任何满足 $k < j_\Delta$ 的 E_k 中都不会出现新事实。

空种子无操作的证明。 假设 $S_\Delta = \varnothing$ 但 $E' \setminus E \neq \varnothing$。首个新事实刻画要求每个新事实的推导都包含 S_Δ 中的一个首个新事实，这就产生了矛盾。因此 $E' \setminus E = \varnothing$。结合加性更新下的单调性， $E \subseteq E'$，即可得出

$$E' = E$$

Wh因为仅有最早的种子并不足够。 位置 j_Δ 只是历史变更中的一个左边界 1 cha。对于每个

$$k \in \mathcal{W}_E(A)$$

if

$$[A \to \bullet\gamma, k]_k$$

如果此前尚未存在于 E 中，那么新激活的产生式 $A \to \gamma \in \Delta P_{\text{act}}$ 会通过预测使该项目本身成为 $E' \setminus E$ 的成员。因此，要实现精确的完整图表等价性，就必须保留所有种子，而不能只保留最早的种子。

A.3 增量 Earley 闭包的证明

本节使用其中引入的关系记号，给出第 3.3 节中省略的推导和正确性论证。

增量预测。 预测关系写作

$$\mathsf{P} = UV$$

其中， $U(k,\ A)$记录了位置 k包含一个等待 A的等待项，而 $V(A,\ q_p^0)$则将一个非终结符映射到其某个可生成产生式的入口状态。语法更新及随后的闭包传播得到

$$V' = V \cup \Delta V, \qquad U' = U \cup \Delta U$$

因此

$$\begin{aligned}\mathsf{P}' &= U'V' \\ &= (U \cup \Delta U)(V \cup \Delta V) \\ &= UV \cup U\Delta V \cup \Delta UV \cup \Delta U\Delta V\end{aligned}$$

由于 $UV = \mathsf{P}$ 并且

$$\Delta UV' = \Delta UV \cup \Delta U\Delta V$$

每个潜在的新预测结论都由以下内容生成

$$U\Delta V \cup \Delta UV'$$

集合成员检查会移除已经具体化的结论。第一项对应于由新激活的产生式激活的已有历史等待项，而第二项则将预测从在增量闭包过程中自身产生的等待项继续传播出去。

增量扫描。 语法更新不会改变词法扫描关系。如果某个扫描结论在更新后变为新增结论，则其期待终结符前提也必然是新增的；否则，对于未改变的已提交前缀，相同的演绎在旧闭包中早已可用。因此，

$$\Delta E_{k+1}^{\text{scan}} = \Delta T_k\, \mathsf{S}_{x_k}$$

其中，S_{x_k} 表示由固定输入符号 x_k所诱导的扫描关系。

增量完成。 令 $W(j, B)$ 表示位于位置 j、等待 B 的项目，并令 $C(j, B)$ 表示起始位置为 j 的已完成 B-成分。旧的完成结论来自

$$J = W \bowtie C$$

With

$$W' = W \cup \Delta W, \qquad C' = C \cup \Delta C$$

所有潜在的新连接均由以下方式生成

$$\Delta J = W \bowtie \Delta C \ \cup\ \Delta W \bowtie C'$$

第一项将新完成的成分与现有等待项进行处理；第二项将新生成的等待项与现有或新生成的完成项进行处理。通过成员检查可以移除任何重复候选项。第 3.3 节所述的键控结构通过将后到达的一侧与已生成、且共享相同 (j, B) 键的事实进行连接，实现了这两种情况。

增量闭包的精确性。 令 E 表示从 S_Δ 开始并反复应用上述增量预测、扫描和完成规则所得到的不动点。那么

$$\widehat{E} = E' = \text{Earley}(G'_{\text{prod}}, x_{<t})$$

可靠性。 每个种子都是在 G'_{prod}下有效的预测结论。随后具体化的每个事实，都是根据在 G'_{prod}下已有效的前提，通过标准 Earley 规则生成的。因此

$$\widehat{E} \subseteq E'$$

完备性。考虑任意

$$f \in E' \setminus E$$

根据首个新事实刻画， f first 的某个推导首次通过 S_Δ 中的一个事实离开旧闭包，而该事实被插入初始工作列表。

沿着该推导的剩余部分进行归纳。如果某个新事实由预测产生，那么它要么是一个初始 $U\Delta V$种子，要么是通过 $\Delta UV'$由新物化的等待项触发。如果它由扫描产生，那么其前提必然是新的，因此会由增量扫描规则传播。如果它由完成产生，那么其两个前提中至少有一个必须是新的；否则旧闭包已经包含该结论。新的完成会通过 $W \bowtie \Delta C$连接，而新的等待项会通过 $\Delta W \bowtie C'$连接。在每种情况下，新结论最终都会被物化。

因此， $E' \setminus E$ 中的每个事实都会被产生，从而得到

$$E' \subseteq \widehat{E}$$

结合可靠性，

$$\widehat{E} = E'$$

对于固定的前缀和有限文法，Earley 项的全集是有限的。由于每个项目仅在首次插入时进入工作列表，因此增量闭包也会终止。

A.4 因子化令牌投影的证明

我们证明第 3.4 节中的命题 2。固定 G' 和 $x_{<t}$，并假设 $E'_{0:t}$ 是 G'_{prod} 下完整且正确的 Earley 历史。对于字节令牌 v，其掩码成员资格定义为

$$M_t(G', x_{<t})[v] = 1 \iff \exists y \in \Sigma^* : x_{<t} b(v) y \in L(G').$$

停止令牌有效，当且仅当 $x_{<t} \in L$（G'）。

引理 1：快速路径的可靠性。 考虑 E'_t 中的活跃生产性循环见证 ℓ，其具有可重复终结符集合 Σ_ℓ。根据循环保持不变量，从 Σ_ℓ 中消费任意完整词法原子，然后执行 Earley 闭包，便可在所得位置重新建立相应的循环项目。因此，直接接受集合可以表示为

$$\mathcal{V}_{\mathrm{acc}} = \bigcup_{\ell \in \mathcal{L}_t} \{v \in \mathcal{V}_{\mathrm{closed}} \mid b(v) \in \Sigma_\ell^*\},$$

其中， $\mathcal{L}_t$ 是经认证的活跃循环见证集合；当不存在此类见证时， $\mathcal{V}_{\mathrm{acc}} = \varnothing$。

对于任意 $v \in \mathcal{V}_{\mathrm{acc}}$，选择一个见证循环 ℓ，并将 b（v）分解为完整词法原子。从其循环项目出发，每个原子都可以通过扫描（Scan）进行消费，随后执行相应的完成（Complete）和 Earley 闭包步骤，从而在下一个位置重新建立循环项目。对 b（v）中的原子进行归纳可知，在整个令牌处理完毕后，所得图表仍保留一个可行的循环状态。由于该循环见证具有生产性，并且属于 G'_{prod} 中的起始可达推导，因此该状态可以有限地完成为一个句子。因此，

$$M_t(G', x_{<t})[v] = 1.$$

反过来，对于任意 $v \in \mathcal{V}_{\mathrm{rej}}$，条件 $b(v)_0 \notin \Sigma_{\mathrm{first}}(E'_t)$ 意味着没有 Earley 项可以扫描其第一个符号。由此得到的图表为空，也没有任何延续能够完成一次推导。因此，

$$M_t(G', x_{<t})[v] = 0.$$

引理 2：推测式字典树推导的等价性。 对于深度为 d的字典树节点 u，令 s_u =str(u)。我们通过对 d 进行归纳，证明其推测状态满足

$$E_u = \mathrm{Earley}(G'_{\mathrm{prod}}, x_{<t} s_u)_{t+d}.$$

对于根节点，这直接源于对完整历史状态 $E'_{0:t}$ 的复用。对于边 $u \xrightarrow{\sigma} w$，假设在 u 处等式成立。追加 σ 并应用扫描、完成和预测，

因此，完整的 Earley 闭包会生成与从头解析 $x_{<t}s_u\sigma$ 得到的图表完全相同的结果。

共享历史表示保留了这一等价性：所有分支都读取不可变的历史图表，而分支特定推导则维护在各自根到节点路径上的覆盖层中。后续 Complete 操作所需的全部事实，均从共享历史状态或活动分支的覆盖层中获取。因此，同级分支无法将事实引入彼此的状态，并且共享同一个字典树前缀不会改变 Earley 推导的结果。

此外，在完全闭包后剪枝那些不包含任何期待终结符项的节点的后代是精确的，因为每个真后代都必须至少再消耗一个终结符，而从这种状态出发这是不可能的。

引理 3：精确的词元有效性刻画。 对于一个经过完全求值的字典树叶节点 u_v，其对应于 $v \in \mathcal{V}_{\text{unc}}$，令

$$\text{Live}(E_{u_v})$$

当且仅当其图表包含期待终结符的 Earley 项或已完成的开始产生式项 $[S' \to S\bullet, 0]$ 时成立。

由于字典树推导会执行完整的 Earley 闭包，期待终结符项会揭示下一个可能的终结符。由于 G'_{prod}中的剩余语法上下文具有生成性，因此它允许有限的终结符补全。反之，任何导向句子的非空延续，在闭包后都必定包含期待终结符项。空延续恰好在已完成的开始产生式项存在时才可能。因此，

$$\exists y \in \Sigma^* : x_{<t}b(v)y \in L(G') \iff \text{Live}(E_{u_v}).$$

由此可得

$$\Pi(E'_{0:t}, \mathcal{V}_{\text{unc}}) = \{v \in \mathcal{V}_{\text{unc}} \mid \text{Live}(E_{u_v})\}.$$

对于停止令牌 $v \in \mathcal{V}_{\text{stop}}$，

$$\begin{aligned} v \in M_t^{\text{stop}} &\iff [S' \to S\bullet, 0] \in E'_t \\ &\iff x_{<t} \in L(G'). \end{aligned}$$

命题 2 的证明。 根据引理 1，快速接受集合和快速拒绝集合能够正确地分类各自的令牌。根据引理 2 和引理 3，推测式字典树推导能够将每个不确定令牌准确地分类为与完整 Earley 解析相同的结果。仅当已提交前缀已经是完整句子时，才会接受停止令牌。

由于

$$\mathcal{V}_{\text{byte}} = \mathcal{V}_{\text{acc}} \sqcup \mathcal{V}_{\text{rej}} \sqcup \mathcal{V}_{\text{unc}},$$

字节令牌词汇表与停止令牌词汇表互不相交，因此它们的并集得到

$$M'_t = M_t(G', x_{<t}).$$

至此，证明完成。

A.5 复杂度分析

设 $t = |x_{<t}|$ 为已提交前缀长度，$|Q| = \sum_{p \in P'}(1 + |\text{RHS}(p)|)$ 为 G' 下带点文法状态的数量。由于图表去重会将每个新推导出的事实在 $\Delta E = \bigcup_{k=j_\Delta}^{t} \Delta E_k$ 中至多一次插入工作列表，算法 1 的运行时间与输出规模相关

$$O\Big(|\Delta P_{\text{act}}| + \sum_{(q,o)_k \in \Delta E} \big(d_{\max} + |\texttt{WaitAdv}[(o,\cdot)]| + |\texttt{CompletedByStart}[(k,\cdot)]|\big)\Big),$$

其中，$d_{\max}$ 是每个非终结符的产生式候选的最大数量。当 $S_\Delta = \varnothing$时，历史更新工作量可降至 $O(|\Delta P_{\text{act}}|)$；当更新从前缀起始位置触发全局推导时，对于任意二义性上下文无关文法，最坏情况下的闭包时间为 $O(|Q|t^3)$，而 $O(|Q|t)$

对于每个带点状态具有有界数量活动起点的语法，保留持久化图表和反向索引在最坏情况下需要 $O(|Q|t^2)$的空间，而在活动起点有界时需要 $O(|Q|t)$的空间；词汇集合 $\mathcal{I}[\sigma]$ 和 $\mathcal{V}_{\text{closed}}$ 占用 $O(|\Sigma| \cdot |\mathcal{V}|)$的静态内存，与 t和 $|Q|$无关。

B 算法与实现

B.1 事件驱动的增量更新

正文中的算法 1给出了完整的识别器更新过程。这里我们将介绍用于实现其操作的持久化表示。

对于一个产生式

$$p : A \to X_1 \cdots X_m$$

GramDelta 会分配稳定的点状态标识符

$$q_p^0, \ldots, q_p^m$$

一个 Earley 事实存储为

$$(q_p^d, o, k)$$

其中， o 是起点位置， k 是当前图表位置。运行时语法添加会在固定大小页面中追加新的产生式和点状态元数据，而不会复制未受影响的页面；与先前已物化事实相关联的标识符永远不会重新编号。

除了请求局部图表成员关系之外，GramDelta 还维护三个反向索引

$$\texttt{HistWait}[A]$$

$$\texttt{WaitAdv}[(k, A)]$$

$$\texttt{CompletedByStart}[(k, A)]$$

HistWait$[A]$ 存储包含至少一个等待 A 的项目的去重图表位置。WaitAdv$[(k,\ A)]$ 存储相应的等待项本身，而 CompletedByStart$[(k,\ A)]$ 存储起点为 k 的已完成 A 成分。后两个结构实现了增量 Complete 连接的两个方向：新的等待项查询已物化的完成项，而新的完成项查询已物化的等待项。

所有派生事实都经过单一的 INSERT操作。对于候选事实 f，INSERT 会检查请求局部图表成员关系。如果 f 已存在，则不再执行任何后续工作。否则，f 会被物化，并恰好一次放入工作列表。算法 1 中的工作列表处理随后更新相关反向索引，并触发预测、扫描或完成后继状态。

对于新激活的产生式 $A \to \gamma$，种子构造仅查询 HistWait$[A]$。此路径无需扫描历史图表。新产生的等待项和完成项随后通过持久化索引传播，直到工作列表为空。因此，该实现遵循第 3.3 节所描述的依赖驱动更新，而不是从受影响的位置重新执行已提交的前缀。

B.2 因子化令牌投影的实现细节

对于使用相同分词器的请求，GramDelta 会共享静态的字节级分词器字典树和预先计算的 64 位打包词汇位集。在分词器初始化时，每个字节 $\sigma \in \{0, \ldots, 255\}$ 都会映射到一个令牌包含位集 $\mathcal{I}[\sigma]$，而 $\mathcal{V}_{\text{closed}}$ 则标识字节序列由完整 UTF-8 码点组成的令牌。包含不完整多字节字符片段的令牌会绕过批量循环接受，并转入精确的字典树推导。

For 对于满足循环保持不变量的活动循环，GramDelta 使用 64- 计算 $\mathcal{V}_{\text{acc}}$
位按位
对预先计算的包含集合执行操作，避免逐令牌解析器推导。

为避免在推测性字典树分支之间重复存储历史图表，已提交行 $E'_{0:t}$ 被组织为不可变的 16 行页。每个分支都共享页目录， O的工作量为 (1)。当

推测性完成在历史起点 $o \leq t$插入一项新的完成内容时，会按需分配行级写时复制覆盖层；各分支独立维护位置超过 t 的推测图表。对于较短的前缀 $t \leq 16$，内联单页路径可减少额外开销。

快速过滤和字典树推导阶段共同为每次请求填充单个打包令牌掩码 M_t'，避免为每次更新重新编译令牌掩码缓存，同时确保跨多字节令牌边界的精确性。

C 实验细节

C.1 工作负载和系统配置

我们在轻量级和持续规模下评估两类运行时语法工作负载。

首先，我们构建受控的动态语法，涵盖递归更新、子语法扩展、局部性扫描（受影响历史行的比例为 0%–100%），以及用于与 Pre[3] 进行 CPU比较的兼容 LR(1)测试装置。这些语法族用于测试递归历史传播、休眠规则重新激活，以及广历史交叉区间。

其次，我们评估不断增长的工具模式和声明工作负载：紧凑的 BFCL 工具调用轨迹（512 个 token 的上下文，$K = 4$–32）以及持续演化的大型语法；后者基于 128 个官方 BFCL 工具模式构建，并采用 CodeEvo 声明驱动合成（∼2.8K 个基础产生式、4K–16K 个前缀、$K = 1$–32 次首次见到的更新）。这一渐进式揭示协议由我们的评估引入，并非原始 BFCL 基准测试的一部分。

我们评估了 Qwen3-8B、Qwen3-32B、Granite-4.1-3B 和 Granite-4.1-8B。所有模型均以 BF16 格式在单个 NVIDIA RTX PRO 6000 Blackwell GPU 上运行，并在统一的进程内 KV 缓存解码框架中执行，使用相同的模型权重、分词器和单次预填充解码配置。

C.2 首次出现更新协议

每条轨迹都从初始语法 G_0 开始，并包含一系列运行时语法增量

$$\Delta G_1, \ldots, \Delta G_m$$

在事件 i之前，后端只能访问当前语法 G_{i-1}。未来的语法片段不会预先注册或编译。在揭示事件发生时， ΔG_i 首次变得可见，活动语法变为

$$G_i = G_{i-1} \cup \Delta G_i$$

更新在当前令牌提交后进行，因此所有方法都针对同一历史前缀 $x_{<t}$处理相同的语法变更。

GramDelta 将 ΔG_i 直接应用于其持久化的请求局部识别器状态。对于同引擎的 Fresh、XGrammar、llguidance 和 Pre[3], ，我们实例化与 G_i对应的后端状态，并重新处理相同的已提交前缀。这条重建并重放路径描述的是它们在我们的首次遇见后延迟更新协议下的处理方式，而不是对这些系统的一般性描述。

更新延迟的测量区间从 ΔG_i 对约束后端变得可见时开始，到更新后的令牌掩码可供下一解码步骤使用时结束。模型前向执行不计入这段事件级时间，而约束状态维护、必要的前缀重新处理以及令牌投影则计入。每个后端都会在正式测量前单独预热，且不会接触属于未来正式事件的语法片段。

主要比较采用共享令牌协议，使所有后端都遵循相同的已提交令牌序列。

令牌策略对齐。 对于 GramDelta、Fresh 和 XGrammar 掩码比较，我们排除空载荷特殊令牌，并且仅在接受状态下允许配置的停止令牌。在共享令牌轨迹上，llguidance 执行其公开的无强制路径（已在掩码差异资格为零的情况下验证），而原生快进解码则在附录 D 中单独报告。

D 扩展实验结果

本附录提供第 4 节所引用的完整跨模型和跨工作负载测量结果，随后分析跨领域行为、编译器缓存、模型规模交互和资源权衡。表3报告了渐进式工具模式发现任务（BFCL、8K 前缀、 $K \in \{16, 32\}$）上的四模型图形处理器结果，表4报告了声明驱动的代码合成任务（CodeEvo、8K 前缀、 $K = 32$）上的相应四模型图形处理器结果。表5总结了四模型静态无更新对照结果（$K = 0$）。表6分解了两个工作负载中的首次出现更新路径。

表3：BFCL 上四种模型的完整图形处理器结果（8K 前缀， $K \in \{16, 32\}$），包含 95% 自助法置信区间和第95百分位更新延迟。

K	模型	方法	更新 p50 [CI] (毫秒) ↓	更新率 第95百分位 (毫秒) ↓	每输出令牌时间 p50 [CI] (毫秒/令牌) ↓	每输出令牌时间开销 [CI] (%) ↓
16	Qwen3-8B	无约束	–	–	19.41 [19.40, 19.42]	0.0
		XGrammar	493.01 [488.85, 495.61]	518.79	79.51 [79.21, 79.73]	309.6 [308.0, 310.7]
		XGrammar-Cache	636.64 [635.03, 639.35]	716.12	98.08 [97.52, 98.68]	405.3 [402.4、408.3]
		llguidance	49.76 [48.97, 50.63]	57.31	23.93 [23.75, 24.08]	23.3 [22.3, 24.0]
		Fresh	40.09 [39.50, 40.59]	149.43	25.22 [25.20, 25.47]	29.9 [29.8, 31.2]
		GramDelta	**3.67 [3.60, 3.77]**	**4.83**	**20.69 [20.68, 20.72]**	**6.6 [6.5, 6.7]**
	Qwen3-32B	无约束	–	–	61.05 [61.04, 61.07]	0.0
		XGrammar	495.24 [493.26, 498.00]	522.79	116.16 [115.49, 116.39]	90.3 [89.2, 90.6]
		XGrammar-Cache	642.86 [638.61, 646.19]	719.85	135.33 [134.70, 135.79]	121.7 [120.6, 122.4]
		llguidance	51.07 [50.38, 51.54]	56.87	63.66 [63.56, 63.73]	4.3 [4.1, 4.4]
		Fresh	41.41 [40.83, 41.86]	168.34	66.89 [66.82, 66.99]	9.6 [9.4, 9.7]
		GramDelta	**3.76 [3.65, 3.87]**	**4.89**	**62.39 [62.36, 62.41]**	**2.2 [2.2、2.2]**
	Granite-4.1-3B	无约束	–	–	15.84 [15.79, 15.93]	0.0
		XGrammar	538.66 [534.11, 541.77]	566.45	81.86 [81.48, 82.63]	416.8 [413.2, 421.4]
		XGrammar-Cache	731.39 [725.68, 735.98]	790.13	106.63 [105.17, 107.49]	573.2 [563.6, 579.5]
		llguidance	50.72 [49.98, 51.29]	55.02	20.76 [20.68, 20.96]	31.0 [30.4, 32.3]
		Fresh	41.26 [40.71, 41.95]	139.46	22.05 [21.68, 22.32]	39.2 [36.8, 40.9]
		GramDelta	**3.73 [3.59, 3.84]**	**4.97**	**17.20 [17.08, 17.37]**	**8.6 [7.7, 9.5]**
	Granite-4.1-8B	无约束	–	–	19.87 [19.84、20.22]	0.0
		XGrammar	528.97 [525.50, 532.68]	560.58	84.65 [83.90、85.17]	326.0 [317.2, 328.7]
		XGrammar-Cache	717.66 [713.84, 720.14]	790.43	108.54 [108.25、109.21]	446.2 [437.1, 449.4]
		llguidance	52.18 [51.27, 52.82]	57.07	24.60 [24.40、24.88]	23.8 [21.5, 25.1]
		Fresh	42.12 [41.54, 42.80]	156.58	26.22 [26.02, 26.72]	32.0 [29.8, 34.2]
		GramDelta	**3.89 [3.73, 3.97]**	**5.01**	**21.53 [21.31, 21.71]**	**8.3 [6.6, 9.2]**
32	Qwen3-8B	无约束	–	–	19.65 [19.50, 19.73]	0.0
		XGrammar	500.68 [495.20, 506.55]	576.22	124.95 [124.30, 127.10]	536.1 [531.3, 548.3]
		XGrammar-Cache	636.70 [630.20, 644.25]	749.03	156.51 [155.76, 157.65]	696.7 [691.2, 705.4]
		llguidance	49.44 [48.57, 50.79]	58.60	27.08 [26.80, 27.69]	37.8 [36.4、41.1]
		Fresh	42.07 [40.74, 44.14]	142.63	28.86 [28.51, 29.40]	46.9 [44.9, 50.0]
		GramDelta	**3.82 [3.68, 3.90]**	**4.93**	**21.17 [21.13、21.30]**	**7.8 [7.3, 8.7]**
	Qwen3-32B	无约束	–	–	61.12 [61.11, 61.13]	0.0
		XGrammar	526.20 [524.03, 528.24]	564.90	163.65 [163.39, 164.03]	167.8 [167.3, 168.4]
		XGrammar-Cache	665.23 [662.82, 668.24]	741.65	196.18 [195.71, 196.93]	221.0 [220.2, 222.2]
		llguidance	51.57 [51.12, 52.05]	57.08	65.60 [65.47, 65.68]	7.3 [7.1, 7.5]
		Fresh	41.92 [41.41, 42.45]	144.73	70.62 [70.25, 70.87]	15.6 [14.9, 16.0]
		GramDelta	**3.75 [3.66, 3.82]**	**4.87**	**62.57 [62.56, 62.60]**	**2.4 [2.3 , 2.4]**
	Granite-4.1-3B	无约束	–	–	15.40 [15.33, 15.51]	0.0
		XGrammar	550.88 [547.78, 554.50]	594.93	133.65 [133.16, 134.20]	768.1 [762.0, 773.1]
		XGrammar-Cache	738.40 [735.73, 740.64]	816.81	176.52 [175.58, 176.90]	1046.6 [1038.2, 1051.7]
		llguidance	49.94 [49.52, 50.39]	54.37	23.88 [23.82, 24.09]	55.1 [54.0, 56.8]
		Fresh	41.33 [41.03, 41.81]	137.40	25.46 [25.22, 25.51]	65.4 [63.5, 66.0]
		GramDelta	**3.80 [3.71, 3.89]**	**4.94**	**16.93 [16.85, 17.01]**	**10.0 [9.3, 10.6]**
	Granite-4.1-8B	无约束	–	–	20.15 [19.87, 20.28]	0.0
		XGrammar	562.08 [557.11, 565.88]	610.07	139.27 [138.48、140.58]	591.1 [583.8, 606.1]
		XGrammar-Cache	750.82 [744.82, 758.24]	835.19	183.20 [180.44, 184.45]	809.1 [790.2, 827.5]
		llguidance	53.27 [52.37, 54.10]	59.22	28.14 [27.92, 28.47]	39.6 [38.1、42.8]
		Fresh	42.95 [42.17, 43.83]	138.66	29.96 [29.43、30.06]	48.7 [45.8, 51.0]
		GramDelta	**3.81 [3.77, 3.91]**	**5.06**	**21.67 [21.39, 21.83]**	**7.6 [6.4, 8.9]**

跨领域一致性（BFCL与 CodeEvo）。对表3和表4的比较表明，在渐进式工具模式发现和声明驱动的代码合成中，排名保持一致。与 BFCL 相比，CodeEvo 上的绝对更新延迟和编译延迟略低（例如，

表4：声明驱动的代码合成在四种模型上的完整图形处理器结果（CodeEvo，8K 前缀，$K = 32$），附有95%自助法置信区间和第95百分位更新延迟。

模型	方法	更新 p50 [CI] (毫秒) ↓	更新延迟 第95百分位 (毫秒) ↓	每输出令牌时间 p50 [CI] (毫秒/令牌) ↓	每输出令牌时间开销 [CI] (%) ↓
Qwen3-8B	无约束	–	–	19.70 [19.42, 19.91]	0.0
	XGrammar	362.07 [359.35, 365.79]	391.88	78.90 [78.54, 80.57]	300.5 [297.5, 307.4]
	XGrammar-Cache	376.24 [373.15, 382.09]	425.65	82.41 [81.90, 83.18]	318.3 [313.3、324.9]
	llguidance	42.40 [41.59, 44.77]	50.43	24.19 [23.97, 24.79]	22.8 [21.4, 25.9]
	Fresh	33.67 [32.95, 34.93]	134.89	26.55 [26.38, 26.78]	34.8 [33.3, 36.8]
	GramDelta	**3.15 [3.07, 3.23]**	**3.71**	**20.74 [20.66, 20.81]**	**5.3 [4.2, 6.8]**
Qwen3-32B	无约束	–	–	61.56 [61.54, 61.58]	0.0
	XGrammar	388.44 [387.31, 389.59]	409.47	118.63 [118.48, 118.87]	92.7 [92.4、93.1]
	XGrammar-Cache	405.72 [404.55, 407.25]	442.58	122.30 [122.11, 122.86]	98.7 [98.3, 99.6]
	llguidance	46.71 [46.16, 47.64]	52.30	64.11 [64.04, 64.20]	4.1 [4.0, 4.3]
	Fresh	35.56 [35.07, 36.03]	147.60	68.87 [68.83, 69.00]	11.9 [11.8, 12.1]
	GramDelta	**3.09 [3.00, 3.16]**	**3.66**	**62.76 [62.72, 62.78]**	**1.9 [1.9, 2.0]**
Granite-4.1-3B	无约束	–	–	15.56 [15.43, 15.79]	0.0
	XGrammar	414.28 [411.65, 417.64]	443.08	85.15 [84.64, 86.09]	447.4 [440.2, 455.4]
	XGrammar-Cache	459.38 [457.00, 462.17]	495.97	93.46 [92.41, 94.12]	500.8 [492.9、505.5]
	llguidance	46.18 [45.60, 46.69]	50.99	21.62 [21.46, 21.84]	39.0 [36.3、40.7]
	Fresh	35.26 [34.86, 35.74]	132.42	23.33 [22.97, 23.44]	49.9 [47.0、51.5]
	GramDelta	**3.19 [3.10, 3.29]**	**3.90**	**17.07 [17.01, 17.24]**	**9.7 [8.5, 11.1]**
Granite-4.1-8B	无约束	–	–	20.08 [20.07, 20.09]	0.0
	XGrammar	410.04 [407.64, 412.43]	433.84	87.79 [87.40, 88.70]	337.3 [335.3, 341.8]
	XGrammar-Cache	455.36 [451.51, 459.12]	493.08	96.22 [95.64, 96.75]	379.3 [376.5, 381.9]
	llguidance	46.68 [46.19, 47.31]	52.44	25.22 [25.14, 25.41]	25.6 [25.2, 26.6]
	Fresh	35.42 [35.14, 35.75]	132.07	27.68 [27.61、27.87]	37.9 [37.5, 38.8]
	GramDelta	**3.31 [3.19、3.36]**	**3.92**	**21.66 [21.64、21.68]**	**7.9 [7.8, 8.0]**

表5：静态无更新 GPU 对照组（$K = 0$，每个单元格进行 30 次配对运行），报告 TPOT 中位数（1K /4K，毫秒/令牌）和配对索引保留开销。

模型	无约束 (1K / 4K)	GD（启用索引） (1K / 4K)	GD（关闭索引） (1K / 4K)	XGrammar (1K / 4K)	llguidance (1K / 4K)	索引税 (1K / 4K)
Qwen3-8B	17.79 / 18.85	18.05 / 19.19	18.03 / 19.19	17.94 / 19.01	18.03 / 19.09	+0.03% / +0.02%
Qwen3-32B	57.10/ 59.50	57.45 / 59.96	57.44 / 59.96	57.32 / 59.71	57.42 / 59.83	+0.01% / +0.00%
Granite-4.1-3B	15.18 / 15.14	15.36 / 15.45	15.34 / 15.41	15.30 / 15.23	15.39 / 15.40	−0.13% / +0.12%
Granite-4.1-8B	18.02 / 19.29	18.26 / 19.64	18.25 / 19.63	18.15 / 19.44	18.25 / 19.54	−0.03% / −0.11%

表6：BFCL 和 CodeEvo 上 Qwen3-8B 的平均更新阶段分解（毫秒）。

工作负载阶段		方法（毫秒）				
		GramDelta	Fresh	llguidance	XGrammar	XGrammar-Cache
BFCL	分派 +主机准备	1.487	38.658	12.256	12.521	12.381
	后端构建/更新	1.480	10.918	23.001	514.676	663.804
	前缀重放	0.000	7.235	20.655	1.927	2.330
	历史查找/闭包	0.001	0.000	0.000	0.000	0.000
	令牌投影	0.098	0.054	0.062	0.008	0.008
	投影后处理/返回	0.728	0.504	0.025	0.004	0.004
	残差	0.022	0.023	0.012	0.064	0.066
	总计	**3.816**	**57.391**	**56.011**	**529.200**	**678.592**
CodeEvo	分派 +主机准备	1.074	34.511	11.471	12.330	12.254
	后端构建/更新	1.149	5.692	21.338	362.190	387.924
	前缀重放	0.000	4.602	13.761	1.389	1.673
	历史查找/闭包	0.004	0.000	0.000	0.000	0.000
	令牌投影	0.165	0.060	0.049	0.007	0.007
	投影后处理/返回	0.663	0.341	0.021	0.004	0.003
	残差	0.028	0.022	0.012	0.062	0.056
	总计	**3.083**	**45.228**	**46.652**	**375.981**	**401.918**

在 $K = 32$，GramDelta 为 3.09–3.31 毫秒，而 XGrammar 为 362.07–414.28 毫秒；相比之下，GramDelta 和 XGrammar 分别为 3.75–3.89 毫秒和 500.68–562.08 毫秒，因为新声明的标识符和类型化成员签名附加到局部化表达式非终结符上，而不是引入嵌套的多参数 JSON 对象模式。在 CodeEvo 上的全部四个模型中，同引擎 Fresh 仍是单事件更新延迟最快的重构基线（33.67–35.56 毫秒），而 llguidance 的重构 TPOT 最低（开销为 4.1–39.0%）；GramDelta 则同时优于二者，其中位更新速度比 Fresh 快 10.70–$11.50\times$，额外 TPOT 开销比 llguidance 低 53.08–76.82%（比 XGrammar 低 97.66–98.24%）。此外，Fresh 的 p95 更新尾延迟较长（CodeEvo 上为 132.07–147.60 毫秒，BFCL 上为 137.40–168.34 毫秒），因为生成过程中越靠后的更新需要重放越长的前缀；而 GramDelta 将 p95 更新延迟限制在 CodeEvo 上的 3.66–3.92 毫秒以及 BFCL 上的 4.83–5.06 毫秒。

持久化编译器缓存的影响。 在表3和表4的所有模型与工作负载中，启用 XGrammar 的持久化跨语法编译器缓存（XGrammar-Cache）所产生的延迟均高于未缓存的 XGrammar（例如，在 CodeEvo 上为 376.24–459.38 毫秒，而未缓存时为 362.07–414.28 毫秒；在 BFCL 上为 636.70–750.82 毫秒，而未缓存时为 500.68–562.08 毫秒，位于 $K = 32$）。在首次出现协议下，每个复合语法 $G_i = G_{i-1} \cup \Delta G_i$ 都包含此前未见过的产生式，并且无法命中完整语法缓存条目，同时仍会产生规范哈希和缓存插入开销，这表明跨语法缓存有利于重复结构，而不是首次出现的运行时新增内容。

更新阶段分解。 表6对两个工作负载下的首次出现更新路径进行了分解。GramDelta 在 BFCL 和 CodeEvo 上分别以 3.816 毫秒和 3.083 毫秒完成更新，且无需重放已提交前缀。其余成本主要分布在分派与主机准备（1.487/1.074 毫秒）以及后端更新（1.480/1.149 毫秒）之间，而令牌投影与投影后处理合计仅占 0.826/0.828 毫秒。相比之下，Fresh 和 llguidance 会产生大量前缀重放成本（4.602–20.655 毫秒），而后端构建/更新占 XGrammar 总更新延迟的 96.3%–97.3%。启用持久化编译器缓存并不会降低这种首次出现的构建成本。这些结果表明，GramDelta 的优势主要来自避免已提交前缀重放和完整后端重构，而不仅仅是加速令牌投影。