软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(5):2103−2130 [doi: 10.13328/j.cnki.jos.007484] [CSTR: 32375.14.jos.007484]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

LLM-Extractor: 基于大语言模型的软件配置间约束提取方法*

张添翼[1,2], 周 彤[1,2], 张晨曦[3], 彭 鑫[1,2]

[1](复旦大学 计算机科学技术学院, 上海 200438)

[2](上海市数据科学重点实验室 (复旦大学), 上海 200438)

[3](西安电子科技大学 计算机科学与技术学院, 陕西 西安 710126)

通信作者: 张晨曦, E-mail: zhangchenxi@xidian.edu.cn

摘 要: 软件配置是软件系统的重要组成部分, 在增强软件功能多样性和灵活性方面具有重要作用. 而随着软件系统越来越复杂, 软件配置项之间复杂的约束关系成为困扰运维人员的问题. 因此研究人员提出了基于不同数据源、使用不同技术的配置约束提取方法, 来识别软件配置之间的复杂约束关系. 然而, 这些方法存在难以应用于多种编程语言、分析规模有限、对高质量有标注数据需求大等多种问题, 针对上述问题提出了一种基于大语言模型的配置间约束提取方法 LLM-Extractor. 该方法包括了配置-功能关联图构建和基于多配置关联子图的配置约束推断两个部分. 在配置-功能关联图构建阶段, LLM-Extractor 借助大语言模型强大的文本理解和分析能力, 从配置文本中识别配置和软件功能相关的实体, 并抽取多种关联关系. 在配置间约束推断部分, LLM-Extractor 在已有配置-功能关联图上搜索多配置关联子图, 并依据关联子图信息引导大语言模型推断配置间约束. 基于多配置关联子图的配置间约束推断方法让 LLM-Extractor 能够提取通过软件功能状态传递的配置约束, 填补了已有方法的空缺, 同时具有对编程语言不敏感、分析规模大的特点. 在 3 个开源软件系统的配置文档上评估了方法的效果, 分析了超过 1400 个软件配置项, 实验结果表明, LLM-Extarctor 的效果相对已有的文本分析方法具有显著提高, $F1$ 分数有至少 43.4% 的提升. 消融实验的实验结果进一步表明, 多配置关联子图对于配置间约束推断方法的效果具有重要的积极影响.

关键词: 软件配置; 软件配置约束; 大语言模型

中图法分类号: TP311

中文引用格式: 张添翼, 周彤, 张晨曦, 彭鑫. LLM-Extractor: 基于大语言模型的软件配置间约束提取方法. 软件学报, 2026, 37(5): 2103–2130. http://www.jos.org.cn/1000-9825/7484.htm

英文引用格式: Zhang TY, Zhou T, Zhang CX, Peng X. LLM-Extractor: LLM-based Constraints Extraction Method Among Software Configurations. Ruan Jian Xue Bao/Journal of Software, 2026, 37(5): 2103–2130 (in Chinese). http://www.jos.org.cn/1000-9825/7484.htm

LLM-Extractor: LLM-based Constraints Extraction Method Among Software Configurations

ZHANG Tian-Yi[1,2], ZHOU Tong[1,2], ZHANG Chen-Xi[3], PENG Xin[1,2]

[1](School of Computer Science, Fudan University, Shanghai 200438, China)

[2](Shanghai Key Laboratory of Data Science (Fudan University), Shanghai 200438, China)

[3](School of Computer Science, Xidian University, Xi'an 710126, China)

Abstract: Software configuration is a crucial component of software systems and plays an important role in enhancing the diversity and flexibility of software functionalities. As software systems become increasingly complex, the intricate constraint relationships between configuration options present a significant challenge for system administrators. To address this, researchers have proposed various

* 基金项目: 国家重点研发计划 (2023YFB4503805); 陕西省自然科学基础研究计划 (2025JC-YBQN-851)
收稿时间: 2024-12-11; 修改时间: 2025-03-17, 2025-04-17; 采用时间: 2025-06-06; jos 在线出版时间: 2025-11-05
CNKI 网络首发时间: 2025-11-06

constraint extraction methods based on different data sources and techniques to identify complex relationships between configurations. However, these methods face several limitations, such as limited applicability across multiple programming languages, constrained analysis scale, and a heavy reliance on high-quality annotated data. To overcome these issues, this study proposes LLM-Extractor, a configuration constraint extraction method based on large language models. This method consists of two main components: the construction of a configuration-function association graph and configuration constraint inference based on multi-configuration association subgraphs. In the graph construction phase, LLM-Extractor leverages the powerful text understanding and analysis capabilities of large language models to identify entities related to configurations and software functionalities from configuration documents and extract various types of relationships. In the constraint inference phase, LLM-Extractor searches for multi-configuration association subgraphs on the existing function graph and guides the large language model to infer configuration constraints based on the information within the subgraphs. By inferring constraints based on multi-configuration association subgraphs, LLM-Extractor can extract configuration constraints transmitted through software function states, filling the gap left by existing methods. It is also characterized by its language-agnostic nature and scalability. The effectiveness of this approach is evaluated on configuration documents from three open-source software systems, analyzing over 1 400 configuration options. Experimental results show that LLM-Extractor outperforms existing text analysis methods, with a 43.4% improvement in $F1$ score. Further ablation studies demonstrate the critical positive impact of multi-configuration association subgraphs on the effectiveness of configuration constraint inference.

Key words: software configuration; software configuration constraint; large language model (LLM)

随着软件系统在社会发展各个领域发挥越来越重要的作用, 社会逐渐进入“软件定义”的时代. 软件配置 (software configuration) 作为软件系统的重要组成部分, 在增强软件功能的多样性、提升系统的灵活性和实现定制化方面具有重要作用. 软件配置广泛存在于从软件开发到软件部署、运维的整个生命周期中. 软件开发人员通过设置配置变量和开发特定配置修改接口, 实现了软件系统的可配置化; 软件部署和运维人员能根据不同的实际应用场景修改对应的软件配置, 从而实现功能定制、资源分配、场景迁移等需求. 近年来, 为了应对更加复杂的软件应用场景, 软件系统逐渐朝着高可配置化的方向发展, 软件配置的灵活性也越来越大.

然而, 随着软件系统规模的增大和复杂性的提升, 软件配置的数量不断增加, 软件配置之间的依赖关系越来越复杂. 例如, Squid 6 拥有超过 300 个配置项[1], MySQL 5.7 数据库拥有超过 1 000 个软件配置项[2], openGauss 5.0.0 拥有超过 700 个配置项[3]. 软件配置复杂性的提高导致配置项相关的软件故障愈发频繁, 许多跨国科技公司的软件故障报告表明, 软件配置错误可能会导致严重的生产事故[4–6]. 2019 年, 一项配置错误导致 Facebook 出现了严重的服务宕机, 影响到数百万用户的正常使用[4]. 2024 年 4 月 8 日下午, 腾讯云出现服务故障, 故障发生后, 依赖云 API 提供产品能力的部分公有云服务, 也因为云 API 的异常出现了无法使用的情况, 此次故障一共持续了 87 min, 腾讯云的故障复盘说明指出错误配置导致了这次故障, 故障的根本原因是配置与新版本的接口协议参数不兼容[5]. 2025 年 1 月 13 日, GitHub 因内部负载均衡器的配置错误导致全球服务中断, 停机时间长达 49 min, 部分用户甚至报告停机时间超过 2 h, 此次事件影响了数以百万计的开发者, 事件报告指出此次事故是由于配置错误影响了数据库基础架构内的流量路由, 导致关键服务意外丢失数据库连接[6]. 种种事例表明软件配置错误会导致严重的后果, 而软件配置之间复杂的约束关系是导致配置错误的重要原因[7]. 软件配置间复杂的约束关系显著提高了配置人员下发错误配置的概率[8,9], 如何在软件配置下发前提前获取软件配置之间的复杂约束关系, 并对软件配置进行错误检测, 是软件配置研究领域的重要问题[10].

软件配置约束提取任务作为软件配置研究领域的重要任务, 有大量研究工作基于不同数据来源、应用多种技术手段来提取软件配置约束[10]. 许多研究工作使用软件系统的源代码作为数据源、应用静态程序分析技术来提取软件配置约束[11–17]. 基于静态程序分析的方法分析软件源代码编译过程中产生的抽象语法树 (abstract syntax tree, AST) 和中间表示 (intermediate representation, IR) 等信息, 从而提取配置约束. 此类方法在实际应用中面临语言特性多样、程序分析规模受限等问题, 无法快速适配不同编程语言编写的软件系统, 并且对于大规模软件系统的分析能力有限[10]. 有部分研究工作使用大量的配置样本作为数据源, 应用概率统计相关方法, 提取软件配置约束[18–24]. 这些方法的效果受数据质量的影响较大, 且提取复杂配置约束的能力较差. 有少量研究工作使用软件配置文档、软件日志等自然语言信息作为数据源, 应用自然语言处理 (natural language processing, NLP) 相关方法, 从

文本信息中提取软件配置约束[25–28]. 由于文本信息的结构化程度较差, 处理难度高, 相关研究工作较少, 且依赖人工进一步介入. 同时软件配置文档等文本中包含了大量配置相关的描述, 且文本分析方法在不同软件之间的适配能力强, 因此通过文本分析提取配置约束是当前软件约束提取任务的重要技术路线.

已有的基于文本分析的配置约束提取方法都依赖预定义的约束模板, 只能提取特定几种模式的配置约束. 但是由于不同软件配置文档的用词、语法等写作方式差异很大, 已有方法的可移植性较差, 依赖预定义模板的准确性和全面性. 通过阅读配置文档我们发现, 配置项与软件功能实体、配置项与配置项之间存在复杂的关联关系, 这些关联关系中隐含着配置间约束关系. 已有文本分析方法受限于逻辑分析能力和上下文分析长度, 无法获取这些隐含的配置约束关系. 如果能够构建软件配置和软件功能实体的关联图, 辅以逻辑推断等手段, 我们就能依据更准确的上下文信息提取这些隐含的配置约束关系.

最近两年, 基于大语言模型 (large language model, LLM) 的方法在文本理解[29]、代码理解[30]等任务上取得了良好的效果. 借助思维链 (chain-of-thought, CoT)、场景学习 (in-context learning, ICL) 等方法, 研究人员可以借助 LLM 从非结构化的文本信息中提取出结构化的关键信息, 实现特定领域问题的信息获取和逻辑推断. 因此借助大语言模型的能力, 我们能够在文本中完成实体抽取、关系抽取、事件抽取、共指消解等重要任务, 自动化地构建软件配置-功能关联图.

因此, 为了解决已有基于文本分析的配置约束提取方法存在的问题, 本文提出了一种基于 LLM 的配置约束提取方法 LLM-Extractor. LLM-Extractor 首先对海量的软件配置文档进行预处理, 随后根据每个配置的描述信息初始化软件配置-功能关联图, 生成配置节点和功能实体节点, 并在图中添加配置属性和功能实体状态等信息; 接着通过分析软件配置文档中存在的逻辑关系, 进一步补全关联图中的配置-功能关联关系. 完成关联图的构建后, 本文以配置条件节点为基点, 使用图搜索算法搜索可能存在的多配置关联子图, 并把关联子图作为提示词的上下文, 引导 LLM 推断配置间约束关系. 通过配置-功能关联图构建和配置间约束推断, LLM-Extractor 能够提取配置文档中隐含的配置间约束关系, 并增强显式配置约束关系的提取效果, 同时减少对于领域专家知识的需求.

为了评估 LLM-Extractor 的效果, 我们人工分析了 3 个广泛使用的开源软件系统附带的配置文档, 分析了 1 400 多个配置项, 构建了配置间约束数据集, 其中包含了 306 条配置间约束关系. 基于该数据集, 本文设置了对比实验和消融实验, 实验结果证明了 LLM-Extractor 在配置间约束提取问题上的有效性.

综上所述, 本文的主要贡献如下.

- 提出一种基于大语言模型的软件配置-功能关联图构建方法, 可以获取并存储软件配置与功能的复杂关联关系.
- 提出并实现了一种基于大语言模型和多配置关联子图搜索的配置间约束推断方法, 该方法可以高效提取软件文本中存在的配置间约束, 并弥补了已有方法无法提取功能状态传递式配置间约束的缺陷.
- 设计了一系列实验验证了方法的有效性.

本文第 1 节介绍软件配置约束提取、LLM 提示工程、文本关系抽取的相关工作. 第 2 节介绍本文研究背景和研究动机, 并通过配置文档经验研究, 对研究动机做进一步阐述, 总结已有方法存在的问题并提出改进思路. 第 3 节介绍 LLM-Extractor 的方法实现细节. 第 4 节描述数据预处理和实验设计及实施的细节, 通过多角度的多组实验验证了方法的有效性. 第 5 节对本文做出总结, 并展望未来研究方向.

1 相关工作

本节首先介绍已有的软件配置约束提取方法, 包括了基于静态程序分析的配置约束提取方法、基于配置文件样本学习的配置约束提取方法和基于自然语言文本分析的配置约束提取方法, 然后探讨已有方法的缺陷. 接着, 介绍大语言模型提示工程相关的研究工作, 阐述大语言模型提示工程在文本理解和逻辑推理等方面的应用潜力. 最后介绍将大语言模型应用于文本关系抽取的相关工作, 展示大语言模型在实体识别、关系抽取等知识抽取任务上的能力.

1.1 软件配置约束提取方法

软件配置约束提取是软件配置研究领域的重要问题, 有大量研究工作基于不同数据源、应用多种方法提取软

件配置约束, 包括基于源代码静态程序分析的方法、基于配置文件样本学习的方法和基于自然语言文本分析的方法, 本节将从这 3 个方面分别进行介绍.

1.1.1 基于静态程序分析的方法

静态程序分析方法是分析软件源代码的有效方法, 可以在不动态运行软件的情况下分析软件中与配置项变量相关的控制流和数据流, 而后研究人员通过分析控制流和数据流, 挖掘其中存在的特定模式, 进而提取软件配置约束. Xu 等人[11]最早将静态程序分析方法应用到软件配置约束提取任务中, 他们分析了多个开源软件的源代码, 总结源代码中软件配置的存在和使用形式, 并归纳出软件配置约束种类, 例如配置类型约束、单配置合法取值约束、多配置取值关联约束等, 在此基础上开发了静态程序分析工具 SPEX 提取上述的多种类型的配置约束. Chen 等人[13]使用类似的思路分析了 OpenStack 和 Hadoop 两种软件生态栈中的 16 种软件, 基于 Soot 开发了适用于 Java 和 Scala 软件配置约束提取的自动化工具 cDep. 与 SPEX 不同的是, cDep 着重分析了多个配置项之间存在的依赖关系. Zhang 等人[14]聚焦软件配置影响的代码片段, 通过分析不同配置对相同代码片段的影响, 提取配置间约束关系, 进而检测软件配置中可能存在的“沉默错误”. 文献 [15,16] 重点分析单配置合法性约束, 通过识别和定位源代码中配置读入和加载的模块, 提取单个配置项对应的类型和合法取值. Zhou 等人[17]研究了 C/C++软件源代码中配置项从配置文件加载到程序变量的过程, 开发了自动化配置变量映射工具 ConfMapper, ConfMapper 可以高效地从 C/C++软件源代码中提取配置变量的映射信息.

基于静态程序分析的方法具有可解释性强的优势, 同时也具有较大的局限性. 首先很多商用软件的源代码没有开源, 无法获取. 除此之外, 现代的大规模软件系统往往使用多种编程语言组合开发, 在这种情况下, 程序分析方法面临分析规模受限、语言特性差异较大的问题.

1.1.2 基于配置文件样本学习的方法

生产环境中的配置文件样本是进行软件配置分析的重要数据来源, 有部分研究工作专注于从海量的配置文件样本中挖掘潜在的配置约束. Zhang 等人[18]通过经验研究分析了配置文件样本中的配置约束类型和特征, 总结了一套从样本中提取配置约束的启发式模板, 使用这套模板从配置文件样本中提取软件配置之间、软件配置与系统环境变量之间的约束关系. Chen 等人[19]的方法与 Zhang 等人类似, 使用启发式模板提取配置约束规则, 不过他们更加聚焦 Web 应用程序不同软件组件的配置之间的约束关系. ConfigC[20]和 ConfigV[21]引入了概率统计的方法, 通过分析配置文件样本数据, 挖掘不同配置项的数值特征, 并通过数值特征推断配置类型等属性信息, 最后基于预定义模板提取配置约束. 还有一些研究工作[22,23]从配置文件版本演化数据、版本维护历史数据等版本数据中挖掘配置频繁模式, 进而提取配置约束. ConfEx[24]则应用了文本相似度匹配算法, 依据配置相关的关键字字典, 识别海量配置文件中的配置约束规则.

基于配置文件样本挖掘的方法, 其优势在于不受特定编程语言的限制, 但具有较强的局限性, 一方面这类方法需要大量的历史配置文件样本, 方法效果依赖样本数据质量. 另一方面, 这类方法依赖预定义模板, 需要大量专家知识的预先输入, 效果受预定义模板的质量影响较大.

1.1.3 基于自然语言文本分析的方法

有少数研究工作基于软件配置相关的自然语言文本提取配置约束. ConfTypeInferer 是 Xu 等人[25]开发的根据配置项名称推断配置项类型的工具, 推断依据是事先归纳的配置类型分类树. PracExtractor 是 Xiang 等人[26]开发的配置约束提取工具, Xiang 等人首先挖掘配置推荐类型语句的词频特征和语法结构特征, 通过配置推荐语句筛选和模式匹配两个步骤提取配置推荐规则. ConfInLog[27]基于静态分析方法分析日志打印点, 筛选配置项相关的日志模板, 随后从软件日志包含的文本语料中提取配置约束关系. Mandal 等人[28]的方法思路和 Xiang 等人类似, 但是 Mandal 等人在语句筛选阶段应用了预训练模型 BERT, 使用大量标注数据来训练分类器筛选配置规则相关的语句, 后续依然使用模式匹配方法提取配置约束规则.

由于自然语言信息结构化较差, 分析难度大, 目前的研究相对较少, 且已有方法高度依赖人工预先定义约束模板, 这对研究人员在相关软件领域的知识水平提出了较高要求. 同时, 已有工作只能分析单个配置相关的配置条件, 且对于语句逻辑的分析能力较差, 无法适应多种多样的自然语言表达形式.

1.2 大语言模型提示工程

大语言模型一般是指参数规模巨大、使用超大规模自然语言语料训练的文本生成模型[31]. ChatGPT 问世以来, 生成式人工智能迎来了大发展时期, GPT (generative pre-trained Transformer)[32]、LLaMA (large language model meta AI)[33]、Gemini[34]等模型反复迭代, 能力不断增强. 大语言模型目前被广泛应用于多种自然语言处理任务, 其能力在文本理解[29]、知识获取[32,35]、逻辑推断[30,36,37]和自然语言生成[38]等任务上得到了验证, 除此之外, 大语言模型相较于传统方法也具有更强的适用性和泛化性.

大语言模型相关的研究工作可以被分为 3 种类型: 预训练 (pre-training)[39]、模型微调 (fine-tuning)[40]以及提示工程 (prompting)[41]. 其中预训练指使用大规模的自然语言数据集或者多模态数据集, 训练生成式模型; 模型微调是针对特定领域任务的优化方法, 在不重新预训练模型的情况下, 使用高质量、有标注的特定领域数据集, 对模型进行端到端参数微调, 这种方法在领域问答等方面应用广泛; 提示工程是应用最广泛的方法, 不需要调整模型本身, 只对提示词做优化, 借助大语言模型强大的文本理解和逻辑推断能力, 解决特定任务.

提示工程的核心是提示词的构建, 研究人员会针对特定任务开发特定的提示模板, 随后将任务的具体数据装填入模板中, 引导 LLM 分析问题并生成目标内容. 为了得到最有效的提示词, 研究人员提出了大量的策略来优化提示内容, 其中思维链和少样本策略应用最为广泛. 思维链策略是一种符合直觉的提示策略, 最早由 Wei 等人[36]提出, 该策略以分步渐进式思考的方式, 引导 LLM 分析问题并输出目标内容, 这种策略被广泛运用于多步骤推理问题. 多项研究工作也表明, 即使仅在提示词中添加“让我们一步一步来思考 (let’s think step by step)”, 也能提高 LLM 的推理和生成性能[42]. 少样本策略是指在提示词中加入少量的提示-生成示例样本, 从而提高模型在特定任务的泛化能力. 在这方面有很多研究聚焦不同类型的样本对生成效果的影响. Zhao 等人[43]认为应当随机地添加样本, Su 等人[44]推荐使用类型丰富的样本, 而 Liu 等人[45]则建议使用和测试集语义相同的示例. 针对文本中软件配置约束提取的各个阶段, LLM-Extractor 均使用了 CoT 来引导大模型分布分析并解决问题, 同时使用了多种示例样本来增强提示效果.

1.3 大语言模型与文本关系抽取

LLM 的强大能力也被应用于文本关系抽取, 已有相当数量的工作研究如何使用 LLM 从非结构化的文本数据中抽取实体和关联关系. Trajanoska 等人[46]探索了使用 LLM 自动化抽取关联关系并构建图结构的能力, 研究表明使用 LLM 的方法相较于传统的自然语言处理方法拥有更高的准确性. Kommineni 等人[47]不仅研究了 LLM 在关系抽取、实体抽取等方面的能力, 还尝试使用 LLM 来半自动化地生成知识图谱本体 (ontology), 研究表明 LLM 能够减少构建过程中的人力介入. Carta 等人[48]基于 GPT-3.5 模型开发了一种构建知识图谱的流水线方法, 在特定领域的数据集上获得了良好的效果. 已有研究表明 LLM 在关系抽取问题上具有良好的效果.

2 背景和动机

本节首先介绍研究问题的背景, 然后通过针对软件配置文档的经验研究, 借助实际示例来说明这项研究的动机.

2.1 研究背景

近年来软件系统的规模越来越庞大、应用场景越来越多, 因此与之对应的软件配置项个数也不断增多, 软件配置项之间的约束关系也变得越来越复杂. 例如, 华为主导开发和商用的 openGauss 数据库拥有 700 多个配置项, 通过与华为 openGauss 数据库运维人员的访谈, 我们了解到, 大部分用户生产环境中的数据库集群处于常年运行的状态, 一年可供重启调整的“天窗”机会一般少于 10 次, 而许多软件配置的生效需要重启数据库, 因此软件配置的下发需要尤为谨慎. 但是运维人员往往只对单一配置项的取值范围和作用有所了解, 对多个配置项之间存在的复杂约束关系缺乏了解, 而这些被忽视的复杂约束关系, 可能在配置实践中产生严重的问题.

在这样的情况下, 软件配置约束提取成为软件配置研究的重点领域, 有大量研究使用静态程序分析、样本学习、文本分析等多种方法来提取软件配置约束. 其中使用静态程序分析和样本学习的方法占多数, 由于程序分析方法存在编程语言适用性有限、分析规模有限等问题, 目前仍停留在研究验证阶段; 而基于样本学习的方法需要

提前获取大量配置文件样本, 且分析效果受数据质量影响较大, 难以提取复杂约束关系. 文本分析方法基于软件文档、软件日志、软件注释等自然语言信息, 从中提取配置约束, 这种方法可以快速分析来自不同软件系统的文本, 具有高度的可移植性. 但是由于自然语言的高度非结构化, 分析难度较大, 研究工作较少.

近两年来大语言模型在自然语言理解和逻辑推理上展现出强大能力, 这让我们能更加高效地完成自然语言分析工作, 这为基于自然语言分析的配置约束提取方法提供了新的思路. 因此本文从自然语言分析的角度入手, 借助 LLM 的能力, 研究软件配置约束的提取方法.

软件系统中的自然语言信息主要有以下 3 种.

(1) 软件文档. 软件文档指描述软件系统、组件、功能以及使用方法的书面材料. 它为开发人员、测试人员、用户以及维护团队提供关于软件系统的关键信息, 帮助他们理解、使用和维护软件. 其中软件配置文档是一种描述软件系统如何进行配置的文档, 通常提供详细的配置说明、参数设置及调整指南. 它帮助开发人员、系统管理员和用户正确配置软件, 以确保系统能够按预期运行. 如图 1 所示, 软件配置文档一般以“配置名称: 配置描述和配置提示”键值对的形式存在.

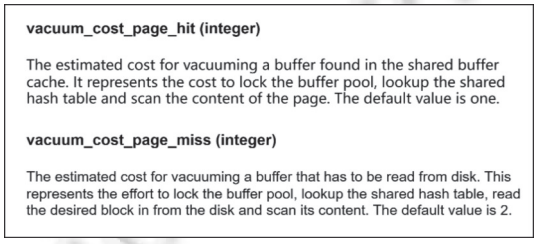


图 1 软件配置文档示例

(2) 软件日志. 软件日志是软件系统自动生成的记录文件, 用于记录系统在运行过程中的各种事件、行为和状态. 软件日志是调试、监控、分析和维护软件系统的重要工具. 它能够帮助开发人员、系统管理员和运维团队了解系统的健康状况、问题出现的原因以及系统的运行状态.

(3) 软件注释. 软件注释是开发人员在代码、配置文件等文件中添加的、供人阅读的说明性文字, 旨在解释代码的功能、意图、逻辑或特殊处理. 注释不会被编译或执行, 它帮助开发者或其他维护人员更好地理解代码.

软件配置文档作为软件配置运维人员的一手资料, 包含的软件配置信息最多、最关键, 因此本文聚焦于软件配置文档, 研究从中自动化提取配置约束的方法.

在阐述本文的研究动机之前, 我们对软件配置约束的概念做出定义.

● 配置约束. 指为了保障软件系统功能模块的正常运行或者软件配置项的正常生效, 单个或多个软件配置项所需要满足的特定约束条件, 这些特定的约束条件必须是明确的、可静态检查的. 配置约束可以分为两种, 单配置约束和配置间约束.

1) 单配置约束. 单配置约束又可以分为配置合法性约束和单配置推荐性约束, 配置合法性约束指的是单个软件配置生效所需要满足的类型、取值范围等必要条件, 单配置推荐性约束指软件文档明确建议将该配置设置为某个范围的取值或者某个特定的取值. 例如在图 2 左侧 fsync 配置项的描述中, fsync 的类型是布尔值、取值范围为 on 和 off, 这属于 fsync 的单配置合法性约束; 而橙色语句说明 fsync 的默认值为 on, 设置为 off 可能会导致不可恢复的数据损坏和系统崩溃, 说明 fsync 被建议设置为默认值 on, 这属于 fsync 的推荐性约束.

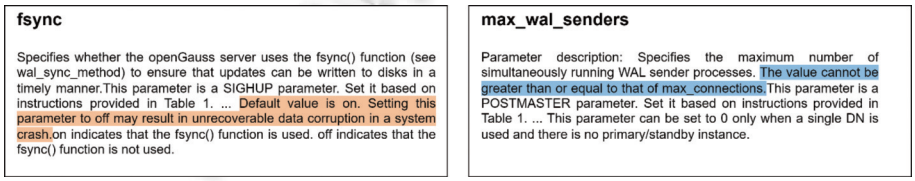


图 2 openGauss 软件配置约束示例

2) 配置间约束. 配置间约束指的是多个软件配置之间具有的关联约束关系, 即一个或多个配置项的设置会影响其他配置项的取值或者生效. 例如图 2 右侧蓝色语句指出 max_wal_senders 应当大于等于 max_connections, 这属于涉及 max_wal_senders 和 max_connections 的配置间约束.

已有工作对于单配置约束的研究已经比较完善, 但是对配置间约束提取的研究尚存在不足, 因此本文主要研究提取多个软件配置间存在的约束关系.

2.2 研究动机

软件文本分析的主要难点在于两个方面: 如何高效分析海量的自然语言信息; 如何从非结构化文本中提取结构化配置约束信息.

对于软件运维人员来说, 软件配置项数量众多, 文档冗长, 因此开发自动化的文档分析工具, 能够大大提高关键配置约束信息的获取效率. 除此之外, 不同文档的行文风格可能存在差异, 表达方式难以统一, LLM 的出现和普及提供了便捷的文本分析工具, 让我们能开发高效的文本分析方法.

为进一步阐述本文研究动机, 我们针对已有文本分析方法泛化性较差、无法挖掘配置间深层次逻辑联系的问题, 对软件配置文档展开了深入的经验研究. 我们人工分析了配置约束相关软件文档的特点, 同时人工分析了文档中存在的配置间约束, 并总结了已有方法无法提取的配置约束类型, 最后根据经验研究结果提出可能的解决方案.

2.2.1 经验研究

为了研究多个软件系统的软件文档中配置间约束的特点, 本文选取了 openGauss 5.0.0[3]、PostgreSQL 12[49]、MySQL 5.7[2]、Squid 6[1]这 4 个被广泛使用的开源软件作为经验研究对象, 人工查看每个软件官网的文档目录结构, 并选择软件配置相关的文档章节作为数据源, 每个软件都拥有数百个配置项, 如表 1 所示.

表 1 配置文档来源和配置数量统计

软件名称	版本	软件配置个数
openGauss	5.0.0	768
PostgreSQL	12	319
MySQL	5.7	1 434
Squid	6	330

我们首先对这些软件的配置文档进行抽样的阅读分析, 随机从配置全集中选取配置项对应的文段阅读分析, 同时阅读章节中可能相关的其他配置项描述, 直至从中分析提取出 100 条配置间约束关系. 对提取这些配置间约束的过程, 我们进行了进一步的分析和总结, 获得了以下发现.

发现 1. 67% 的配置间约束关系无法通过已有研究总结的模板匹配方法提取.

已有方法[26,28]都首先筛选配置设置建议类型的语句, 而后使用正则表达式等模板匹配方法从语句中提取约束规则, 经验研究表明, 100 条配置间约束关系中只有 33 条可以使用这种方法提取, 有 67 条配置约束无法使用这种方法提取. 经过对这 67 个负样本的分析, 我们发现无法提取的原因有以下 3 点.

(a) 44.8% 的负样本对应的配置文本语句中不包含推荐性质的关键词, 例如 recommended、suggested.

(b) 20.9% 的负样本对应的配置文本语句中包含有多个配置条件, 而已有方法无法处理多个配置条件之间的逻辑关系.

(c) 32.8% 的负样本具有已有方法模板未覆盖的表达形式. 由于已有方法通过正则表达式提取对应配置的取值, 如果目标文本的表达方式不能被模板匹配, 就无法被识别为配置取值条件. 例如“You are advised to set this parameter to a value greater than or equal to twice of max_changes_in_memory”中“greater than or equal to twice of max_changes_in_memory”并未被模板成功识别, 导致该条约束未能被成功提取.

发现 2. 47% 的配置间约束无法从单个配置的描述文本中完成提取.

47 条配置约束需要结合多个配置项的描述文本来进行推断, 这些归属不同配置项的文本描述通过特定软件功能实体链接. 图 3 展示了一条配置约束推断的思维链, 研究人员阅读 bgwriter_lru_multiplier 的描述文档时意识

到, bgwriter_lru_multiplier 的配置正常生效需要 background writer 能够正常运行, 从而联想到之前阅读过的 bgwriter_lru_maxpages 描述中有关 background writer 的描述, 如图 3 右侧蓝色部分“When this parameter is set to 0, the background writer is disabled”, 即如果 bgwriter_lru_maxpages 被设置为 0, 则 background writer 处于 disabled 状态, 这与 background writer 正常运行的要求产生了矛盾, 于是研究人员推理出一条配置约束:

(bgwriter_lru_maxpages==0) *LeadTo* NotEffect(bgwriter_lru_multiplier).

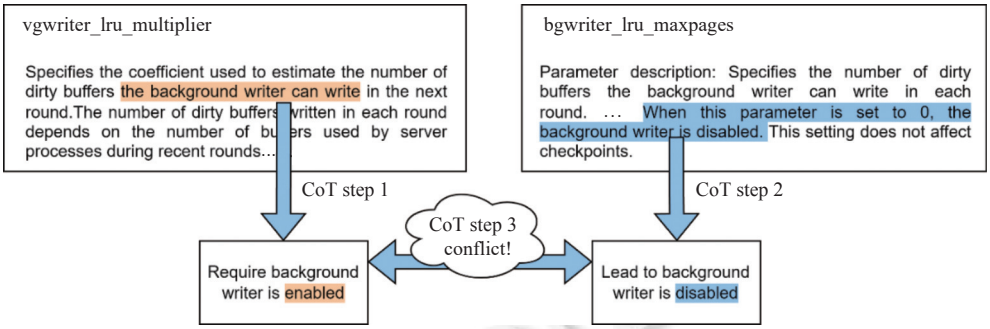


图 3 bgwriter_lru_maxpages 和 bgwriter_lru_multiplier 约束提取思维链

在以上的推理过程中我们发现, 研究人员综合分析两个不同的配置项的描述文本推理出了配置间约束, 而正是 background writer 这一软件功能实体使这两个配置项产生关联. 而在图 4 展示的案例中, 配置间约束展现了更复杂的关联路径. 如图 4 所示, 研究人员阅读配置项 audit_enabled 的描述时得知将 audit_enabled 设置为 off 会导致 auditing function 被关闭; 接下来研究人员阅读配置项 audit_resource_policy 的描述时发现 audit_resource_policy 作用于 audit logs 功能, 当 audit_resource_policy 被设置为 off 时, audit logs 按照最小持续时间作为触发条件, 结合 audit_enabled 的描述, 研究人员认为 audit logs 功能还依赖 auditing function 的开启; 而在 audit_file_remain_time 的描述中研究人员发现 audit_file_remain_time 定义了触发 audit logs 的最小持续时间, 因此 audit_file_remain_time 依赖于 audit logs 将时间作为触发条件, 即“audit logs are preferentially stored by time”. 总结上述一系列发现, 研究人员得出了以下配置间约束:

(audit_enabled==off or audit_resource_policy!=off) *LeadTo* NotEffect(audit_file_remain_time).

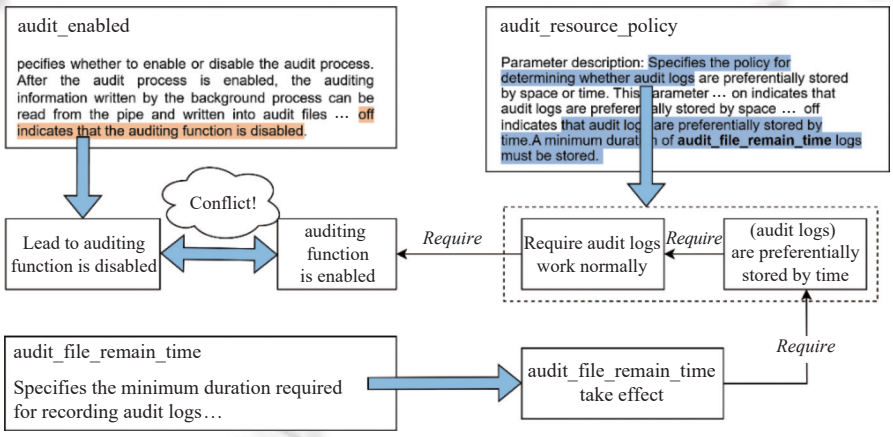


图 4 audit_file_remain_time 约束提取思维链

从以上的推理中可以发现, 该条配置约束涉及软件配置对软件功能状态的影响、软件功能之间的依赖关系、软件配置之间的依赖关系等多种依赖关系, 研究人员通过对这些依赖关系的理解、记忆和组合, 推断出多个配置之间存在的约束关系. 回顾约束推断过程并进行抽象, 我们发现配置间约束推断的依据是一个多种异构节点相互

关联形成的图结构, 而约束关系通过该图结构上的节点和边传递. 对于图 4 所示的示例, 我们对推断过程涉及的配置、功能、条件等要素进行抽象, 构建了如图 5 所示的图结构. 如图 5 所示, 我们将配置间约束推断中涉及的软件对象抽象为几种不同的节点, 即软件配置 (图中灰色椭圆)、软件配置条件 (图中菱形)、软件功能 (图中橙色矩形)、软件功能状态 (图中白色无填充矩形), 并使用 *LeadTo*、*Require*、*Related* 等几种逻辑关联关系将这些节点关联, 形成了多配置相互关联的图结构. 事实上, 研究人员正是在脑海中生成了与图 5 类似的图结构, 并经过思考和推理, 得出了配置间约束规则, 而在这个过程中研究人员需要回忆不同配置文本的描述, 并将这些描述中的关键信息进行整合. 由此我们受到启发, 如果能够通过一个配置-功能关联图将不同配置文本描述中的逻辑关联信息记录下来, 形成结构化的知识, 再借助 LLM 的文本理解和逻辑推断能力, 就能模拟研究人员阅读文本并推断配置间约束的过程. 相比于直接引导 LLM 阅读冗长杂乱的配置文档, 我们认为这种添加针对性上下文的方式能够显著提高配置间约束推断的效果.

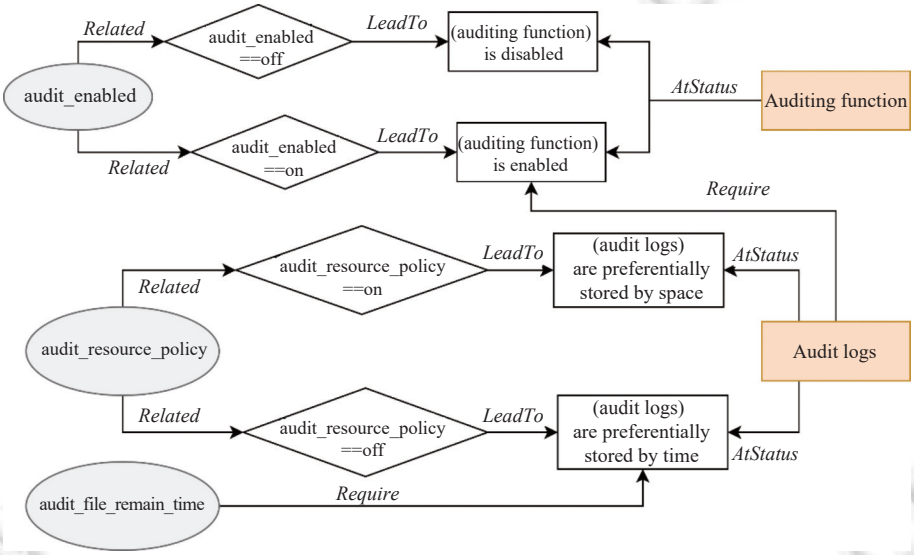


图 5 audit_file_remain_time 约束提取思维链图结构

由此我们先借助 LLM 分析每个配置文档描述的内容, 提取其中包含的配置条件、软件功能及其状态等信息, 再将这些实体关联, 形成配置-功能约束关联图. 基于该图结构, 我们可以找到存在多配置关联的子图, 进而基于子图推断不同配置项之间存在的约束关系.

2.2.2 方法动机

通过上述的经验研究我们发现, 已有方法的缺陷主要在于文本理解能力较差, 无法准确识别软件配置项对应的配置条件, 并提取不同配置条件之间的逻辑关系, 同时, 已有方法未能综合分析不同配置项的文本描述之间的关联关系, 而这些关联关系通过软件功能实体的状态进行传递. 基于这些发现, 我们提出了基于 LLM 的配置间约束提取方法, 借助 LLM 在文本理解分析领域的强大能力, 能够高效识别文本语句中的配置项以及对应的配置条件, 并抽取配置条件之间、配置条件与软件功能实体之间以及功能实体之间的关联关系, 构建出软件配置-功能关联图. 基于该图, 我们可以搜索相互关联的软件配置以及关联子图, 并把关联子图作为上下文提供给 LLM, 从而发挥 LLM 的逻辑分析能力, 推断多个软件配置项之间的约束关系并输出约束规则.

3 方 法

本文提出了基于 LLM 提示工程的软件配置约束提取方法 LLM-Extractor, 图 6 展示了 LLM-Extractor 方法的整体流程. 该方法可以被分为两个主要步骤: 软件配置-功能关联图 (config-function relation graph, CFRG) 构建和

基于 CFRG 的软件配置约束推断. 在 CFRG 构建步骤, LLM-Extractor 输入目标软件的配置文档, 分 3 个步骤构建关联图: 数据获取和预处理 (第 3.1.1 节)、配置-功能关联图初始化 (第 3.1.2 节)、配置-功能关联图补全 (第 3.1.3 节). 在基于 CFRG 的软件配置约束推断中, 输入构建完成的 CFRG, LLM-Extractor 分两个步骤提取配置间约束: 多配置关联子图搜索 (第 3.2.1 节) 和配置间约束推断 (第 3.2.2 节). 本节将详细介绍 LLM-Extractor 的实现细节.

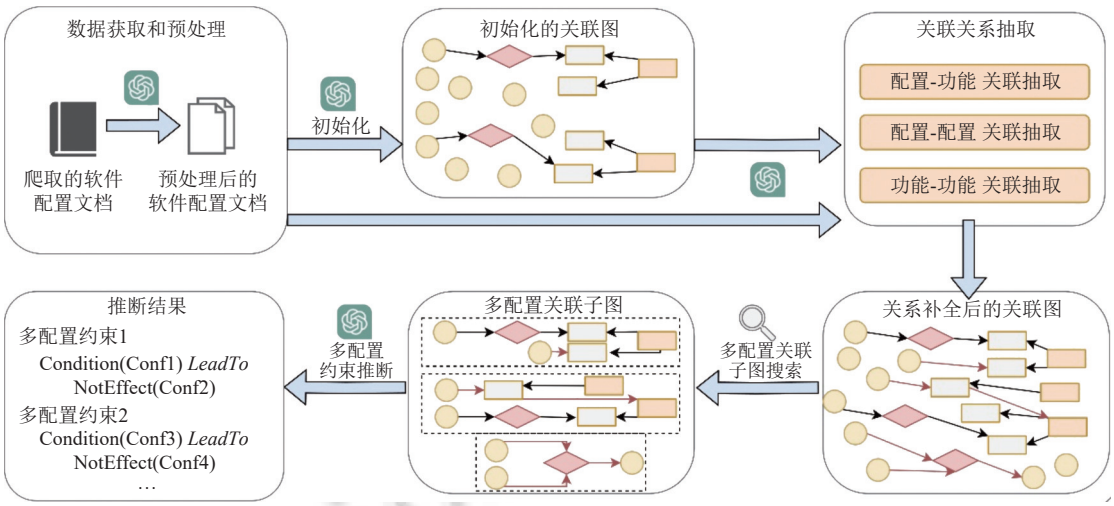


图 6 LLM-Extractor 流程总览

3.1 配置-功能关联图构建

在第 2.2.1 节的经验研究中, 本文发现研究人员在人工阅读文档、搜索配置间约束时, 会在脑海中形成对已阅读配置项的记忆, 这种记忆不仅包括配置项本身的含义, 还包含它们对系统功能和其他配置项的潜在影响. 当研究人员阅读新的配置项时, 这些存储在记忆中的信息会被快速唤醒, 从而辅助他们综合判断新配置项与已有配置项之间是否存在约束关系. 这种认知过程涉及信息的存储、联想和推理, 是人类分析多配置关系的核心方式. 受到这一人工分析流程的启发, LLM-Extractor 在推断配置间约束之前, 首先会对每个配置的描述文档进行深入分析, 并构建一个软件配置-功能关联图 (CFRG). 该图记录了软件配置项与系统功能之间的关联关系, 提供了一种结构化的知识表示形式. 通过构建 CFRG, 不同配置项与功能之间的相互作用能够被清晰地记录下来, 有助于后续的配置约束推断. 在完成 CFRG 的构建后, LLM-Extractor 模拟人工分析中的回忆、思考和推断过程, 基于 CFRG 搜索多配置关联的上下文信息, 借助这些上下文信息推断出潜在的配置约束关系. 通过这一过程, LLM-Extractor 提升了配置间约束推断的准确性和可解释性. CFRG 是 LLM-Extractor 方法的重要数据依据, 下面将进一步介绍 CFRG 的内容以及图定义.

CFRG 是一种用于记录软件配置与功能实体之间控制和依赖关系的关联图, 具体而言, 它涵盖了 3 种主要的关联关系: 1) 配置与功能的关联关系, 这类关系可概括为两种逻辑关系, 即配置控制功能和配置依赖功能, 前者体现了配置对软件功能的控制作用, 后者则反映了软件功能对配置生效的约束作用; 2) 多个配置之间的直接关联关系, 这种关系描述了多个配置对软件功能的共同影响, 或者一个配置对另一个配置生效的约束作用; 3) 多个功能之间的关联关系, 这类关系记录了不同功能之间的相互依赖, 一般表现为一个功能的生效依赖于另一个功能的生效. 为了准确记录这些关联关系, 本文定义了配置节点、功能实体节点、配置状态节点、功能状态节点和配置条件节点等图节点概念, 并将上述 3 种关联关系以“节点-边-节点”的形式存储到 CFRG 中, 从而形成一个具备可搜索性的关联图, 为后续的配置间约束推断提供了结构化知识支持.

CFRG 的具体图定义为: 对于一个 CFRG 实例图 G, $G=\langle E,R\rangle$, 其中 E 表示图中节点, R 表示节点之间的关联关系, E 和 R 的类型可以分为多种, 图中节点和边的定义如表 2 所示, 下面将介绍图定义的具体内容.

表 2 CFRG 的实体类型和关系类型定义

类型	类别	定义	定义描述	示例
节点	E_{C}	$E_{\mathrm{C}} \subset allconfigs$	软件配置项: 软件文档中存在对应描述的所有配置项	enable_double_write, wal_sync_method
	E_{CS}	$c \in E_{\mathrm{C}}, sfromtext$ $(e_{\mathrm{Cond}} \rightarrow AtStatus(e_{\mathrm{C}}, e_{\mathrm{CS}})) \rightarrow e_{\mathrm{CS}} \in E_{\mathrm{CS}}$	软件配置状态: 该状态由确定的配置条件导致	wal_sync_method, doesn't take effect(fsync==off)→ *AtStatus*(wal_sync_method, doesn't take effect)
	E_{Cond}	$c_1, c_2, \ldots \in E_{\mathrm{C}}$, $v_1, v_2, \ldots \in Constant$, $\oplus \in [<>\leqslant\geqslant=]$ $e_{\mathrm{Cond}} = c_1 \oplus v_1 \wedge c_2 \oplus v_2 \wedge \ldots \rightarrow e_{\mathrm{Cond}} \in E_{\mathrm{Cond}}$	软件配置条件: 由一个或多个配置相关的值约束组成	enable_double_write==off wal_sync_method==fsync
	E_{F}	$\exists e_{\mathrm{Cond}} \in E_{\mathrm{Cond}}, e_{\mathrm{F}}, e_{\mathrm{FS}}$ from text $(e_{\mathrm{Cond}} \rightarrow AtStatus(e_{\mathrm{F}}, e_{\mathrm{FS}})) \rightarrow e_{\mathrm{F}} \in E_{\mathrm{F}}$	软件功能实体: 配置文档中提到的功能实体, 功能实体的状态应当被明确的配置条件决定	background writer (bgwriter_lru_maxpages==0 → *AtStatus*(background writer, disabled))
	E_{FS}	$\exists e_{\mathrm{Cond}} \in E_{\mathrm{Cond}}, e_{\mathrm{F}}, e_{\mathrm{FS}}$ from text $(e_{\mathrm{Cond}} \rightarrow AtStatus(e_{\mathrm{F}}, e_{\mathrm{FS}})) \rightarrow e_{\mathrm{FS}} \in E_{\mathrm{FS}}$	软件功能实体状态: 配置文档中提到的功能实体, 状态应当被明确的配置条件决定	background writer is disabled (bgwriter_lru_maxpages==0 → *AtStatus*(background writer, disabled))
关系	*AtStatus*	E_{F} *AtStatus* E_{FS} E_{C} *AtStatus* E_{CS}	软件功能 E_{F} 处在状态 E_{FS} 或者软件配置 E_{C} 处在状态 E_{CS}	(background writer) *AtStatus*(is disabled)
	RelatedTo	E_{C} *RelatedTo* E_{Cond}	软件配置 E_{C} 与配置条件 E_{Cond} 相关联	(bgwriter_lru_maxpages) *RelatedTo* (bgwriter_lru_maxpages==0)
	LeadTo	E_{Cond} *LeadTo* E_{FS}	配置条件 E_{Cond} 会导致功能 E_{F} 处在状态 E_{FS}	(bgwriter_lru_maxpages==0) *LeadTo* (background writer is disabled)
	Require	E_{C} *Require* $E_{\mathrm{Cond}} E_{\mathrm{FS}}$	软件配置 E_{C} 生效要求满足配置条件 $E_{\mathrm{Cond}} E_{\mathrm{F}}$ 或者满足 E_{F} 处在状态 E_{FS}	bgwriter_lru_multiplier *Require* (background writer is enabled)

E_{C} 代表当前软件系统的配置项, 软件配置项还具有配置类型和配置合法值两种属性, 一个配置项对应一个配置类型, 但是可以对应多个合法取值, 如表 2 中 wal_sync_method 对应类型是枚举型, 对应合法取值有 open_datasync、fsync 等. E_{CS} 代表软件配置项生效相关的状态, 例如"doesn't take effect""take effect", 又例如"recovery_parse_workers prevails recovery_max_workers", 这些状态由特定的、明确的配置条件导致.

E_{Cond} 代表文档中出现的配置条件, 可能是单个配置项满足的取值条件, 例如 wal_sync_method==fsync, 也可以是多个配置项取值条件析取或者合取的组合, 例如 wal_sync_method==fsync ∧ enable_double_write==on 是由两个单配置条件合取连接成的组合配置条件.

E_{F} 和 E_{FS} 是本文根据第 2.2.1 节中经验研究的结果定义的软件功能实体相关概念, 由于本文的研究问题是配置间约束提取, 而配置约束必须包含明确的、可检查的配置条件 E_{Cond}, 因此我们只关注被 E_{Cond} 控制的软件功能实体以及功能实体对应的状态, 因此 E_{F} 和 E_{FS} 的定义为: 对于一段文本语段, 如果可以从中提取 $e_{\mathrm{Cond}} \in E_{\mathrm{Cond}}$、$e_{\mathrm{F}}$ 和 e_{FS} 满足 e_{Cond} *LeadTo* $AtStatus(e_{\mathrm{F}}, e_{\mathrm{FS}})$, 我们则认为 e_{F} 属于功能实体 E_{F}, e_{FS} 属于实体状态 E_{FS}, 其中 $AtStatus(e_{\mathrm{F}}, e_{\mathrm{FS}})$ 指功能实体 e_{F} 处于状态 e_{FS}. 例如在 bgwriter_lru_maxpages 的描述语句"When this parameter is set to 0, the background writer is disabled"中, 我们可以提取 (bgwriter_lru_maxpages==0) *LeadTo* *AtStatus*(backgroundwriter, disabled), 其中 backgroundwriter 属于功能实体 E_{F}, disabled 属于功能状态实体 E_{FS}.

在实体间关联关系的定义上, R=〈*AtStatus*, *RelatedTo*, *LeadTo*, *Require*〉, 如表 2 关系部分所示, 定义了不同类型实体之间的关联关系.

介绍完构图动机和图定义之后, 本文下面将正式介绍 CFRG 构建的 3 个步骤: 数据获取和预处理、CFRG 初始化和 CFRG 关系补全.

3.1.1 数据获取和预处理

在本文中, 使用软件的配置文档作为构建 CFRG 的数据源. 目前主流开源软件大都提供从 Web 端访问软件配置文档的入口, 有的软件会提供官方文档的 pdf 版本以供下载. 我们使用 Python 语言编写脚本工具从官方网站获取配置文档数据, 针对不同软件的官方网站编写适配器, 爬取配置文档并统一转化为{配置: 描述文本}的键值对形式.

获取的配置文档仍需要预处理来用于进一步分析. 本文借助 LLM, 通过 3 方面的预处理来提高数据质量: 1) 主语补充, 这一步补充文档语料中缺失的配置主语, 将配置项名称作为主语显式地补充在语句中, 如图 7 所示, 在配置项 enable_memory_limit 的描述中, Specifies 缺少 enable_memory_limit 作为主语, 此处引导 LLM 将主语补充完整; 2) 共指消解 (coreference resolution), 该步将文本中不同的词语或短语, 识别为它们所指的同一个对象或实体, 例如在图 7 中, enable_memory_limit 被 it 和 this parameter 指代, 经过替换后, 新的语句如图 7 右图蓝色部分所示; 3) 配置标注, 这一步将语句中的配置项使用特定标识符标识出来, 这样便于引导大模型识别语料中的配置项关键词, 提升大模型处理语料的效果, 如图 7 所示, enable_memory_limit 被使用<<>> 标注, 便于后续大模型将<<>> 中的内容识别为配置项关键词.

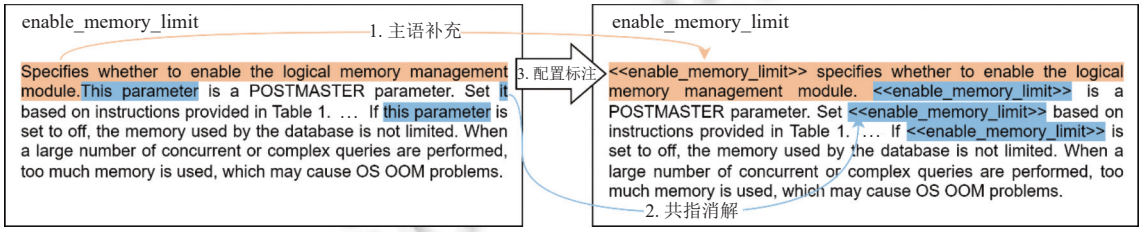


图 7 文档数据预处理示例

3.1.2 配置-功能关联图初始化

LLM-Extractor 首先遍历每个配置项的文本描述, 从中分析配置项基本属性信息并抽取软件功能实体, 构建 CRFG 的骨架, 完成关联图初始化.

(1) 类型抽取和合法取值抽取

LLM-Extractor 首先引导 LLM 分析文本描述并判断配置项的类型, 而后根据不同的类型抽取配置项的合法取值. 本文结合已有工作的配置分类方法以及第 2.2.1 节中经验研究的结果, 将配置分为 4 类: 布尔、数值、枚举和复杂字符串, 本文重点考虑前 3 种类型. 配置项类别以及对应的合法值抽取提示词如表 3 所示.

表 3 配置项分类和对应合法值抽取提示词

配置项类别	举例	合法值抽取提示词
Boolean	enable_double_write: on/off	If {config} belongs to Boolean type, please extract the value set for True as well as for False. Ex: True: [on, enabled] False: [off, disabled]
Numerical	bgwriter_lru_maxpages: 0–1 000	If {config} belongs to Numerical type, please extract the legal value range for {config}. Ex: int(1, 1 000)
Enum	wal_sync_method: open_datasync/fsync…	If {config} belongs to Enum type, please extract the legal value for {config}. Ex: [open_datasync, fsync, fsync_writethrough, open_sync]
ComplexString	krb_srvname: {customed name}	—

(2) 软件功能和状态实体抽取

LLM-Extractor 引导 LLM 分析文本抽取文本中存在的软件功能实体和状态实体. LLM-Extractor 逐句输入配置文档, 判断语句中是否存在满足 e_{Cond} *LeadTo AtStatus* $(e_{\text{F}}, e_{\text{FS}})$ 逻辑关系的自然语言表述, 如果存在, 则从文本中抽取出对应的 e_{Cond}、e_{F}、e_{FS}, 并输出关联关系, 最后将解析出的关联关系补充到 CFRG 中.

由于使用 LLM 提取软件功能实体时存在稳定性较差、依赖文档本身写作风格的问题, 本文对提取出的 E_{F} 集合进行实体消歧. 本文使用预训练的文本嵌入模型 bge-large-en 计算不同 E_{F} 实体的向量表示, 随后通过计算向量表示之间的相似度判断实体相似度 s, 若 s 高于设定的相似度阈值, 则认为该两个实体指向同一个 E_{F}. 完成实体消

歧后, LLM-Extractor 还会检查每个 E_{FS} 实体, 添加 E_{FS} 对应的相反状态实体, 这是由于配置文档总是倾向于描述功能在特定条件下被关闭或者禁用, 而缺乏类似开启、启用的描述. 例如在图 8 中, LLM-Extractor 在配置 bgwriter_lru_maxpages 的文本表述中提取出 E_F 实体“the background writer”和 E_{FS} 实体“(the background writer) is disabled”, 此时 E_{FS} 实体“(the background writer) is disabled”缺少与之对应的相反状态实体“(the background writer) is enabled”, 这会对后续 E_C-E_F 关系抽取造成负面影响. 为了解决这个问题, LLM-Extractor 使用一种基于关键词的启发式方法, 检查 E_{FS} 实体是否具有 disabled、blocked、not used 等状态的关键词, 再添加相应的相反状态.

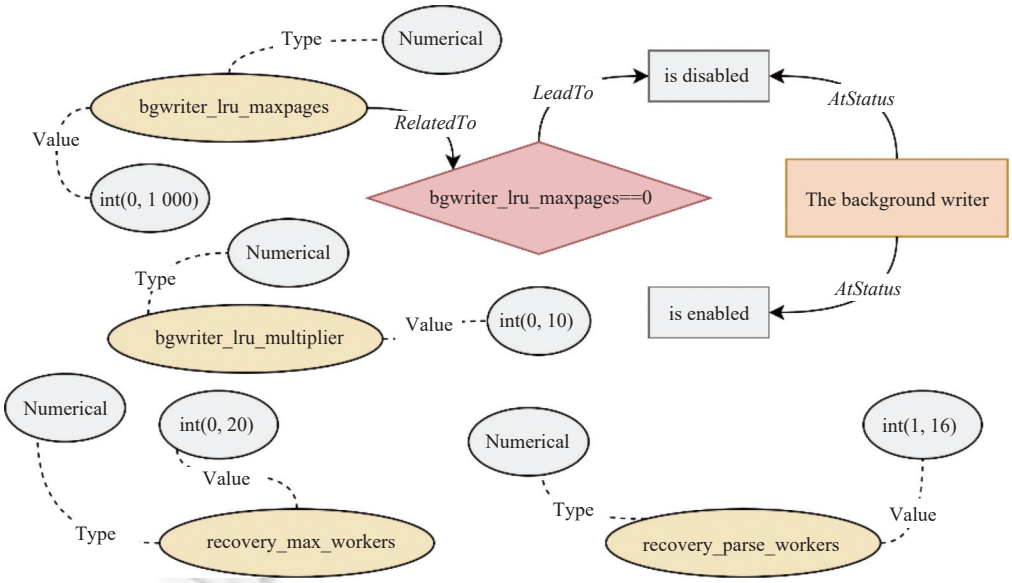


图 8 CFRG 初始化示例

完成上述处理后, 我们就完成了 CFRG 的初始化, 图 8 展示了一个 CFRG 初始化之后的部分节点和关联关系, 从图 8 可以看出, LLM-Extractor 在 CFRG 初始化阶段识别了配置项 bgwriter_lru_maxpages 对功能 the background writer 的影响, 并抽取了相应的实体和关联关系, 而对于配置项 recovery_max_workers 和 recovery_parse_workers, LLM-Extractor 仅抽取了与配置实体对应的类型和合法取值. 在后续的 CFRG 关系补全步骤中, LLM-Extractor 将抽取更多的关联关系添加到 CFRG 中, 作为配置间约束推断的数据基础.

3.1.3 配置-功能关联图补全

在完成 CFRG 的初始化之后, LLM-Extractor 构建了目标软件的配置-功能骨架图, 还需要更加细粒度地分析配置文档, 挖掘更多关联关系来链接不同的 E_C 和 E_F 节点, 为后续的多配置关联子图搜索提供数据基础. 这一步具体可以分为 3 部分: E_C-E_F 关系抽取、E_C-E_C 关系抽取和 E_F-E_F 关系抽取, 接下来将详细介绍.

(1) E_C-E_F 关系抽取

这一步的目的是抽取软件配置项 E_C 对软件功能 E_F 特定状态的依赖关系, 即 e_C *Require AtStatus* (e_F, e_{FS}). 由于在 CFRG 初始化阶段, LLM-Extractor 已经提取了每个软件配置项 E_C 对于软件功能实体 E_F 状态的边际影响, 即 E_{Cond} *LeadTo* E_{FS}, 因此提取 E_C 对软件功能 E_F 的依赖关系可以将不同的 E_C 通过同一个 E_F 链接起来, 形成多个配置之间的关联链路.

LLM-Extractor 首先识别语句中的软件功能实体 E_F. 识别 E_F 而不是直接识别 E_{FS} 的原因是 E_F 表述的稳定度相对较高, 但是 E_{FS} 往往有多种表达方式, 而且有可能缺乏显式表达, 例如 bgwriter_lru_maxpages 的描述“bgwriter_lru_maxpages specifies the number of dirty buffers the background writer can write in each round”中隐含了 *AtStatus*(background writer, is enabled) 作为必要条件. 本文使用词向量匹配的方法搜索语句中存在的功能实体, 即使用滑动窗口在语句的词序列上滑动, 计算窗口内短语与功能实体 E_F 的语义相似度, 进而判断是否匹配. 这样的

方法存在两个问题: ① 文本数据庞大, 逐个计算文本向量并计算相似度会消耗大量时间, 效率比较低; ② 功能实体的短语长度方差很大, 固定窗口长度会显著影响匹配效果.

为了解决这两个问题, LLM-Extractor 使用可变滑动窗口结合向量搜索的方法进行 E_F 识别. 方法具体流程如算法 1 所示. 在第 5–7 行 LLM-Extractor 首先加载 bge-large-en 模型并使用 faiss[50]生成所有 E_F 的索引, 使用 faiss 索引搜索相近词向量, 从而解决上述问题 ①; 随后在第 8 行 LLM-Extractor 计算所有 E_F 实体的平均长度, 作为初始窗口长度; 接着在第 9–15 行 LLM-Extractor 使用初始窗口滑动遍历文段, 计算整个文段和所有 E_F 实体的向量欧氏距离之和, 选出距离最近的 k 个实体再次计算平均长度作为窗口大小. 这一步的目的是减小窗口大小和目标实体长度差距过大导致的误差, 从而解决上述问题 ②; 最后在第 17–21 行, LLM-Extractor 使用新的窗口再次滑动遍历文段, 利用 faiss 索引搜索和每个窗口向量欧氏距离最小的 E_F 实体, 若该欧氏距离小于预先设定的阈值, 则把该匹配加入返回值集合中. 通过算法 1, LLM-Extractor 完成了文段中的 E_F 实体识别, 并记录功能实体匹配的位置.

算法 1. E_F 实体识别算法.

输入: 软件功能实体集合 E_F, 预训练的文本嵌入模型 *bge*, 配置文本段集合 P;
输出: 文本窗口和匹配到的目标实体 e_F 的对应关系列表 *Matches*.

```
1. function EntityIdentification (E_F, bge, P)
2.     Let Matches be match list of (p, window, e_F)
3.     Let Embeddings be embedding list of E_F
4.     Model ← FlagModel(bge)
5.     Embeddings ← Model.encode(E_F)
6.     index ← faiss.IndexFlatL2(dimension) //选择欧氏距离作为搜索方法, 生成 faiss 索引
7.     index.add(Embeddings)
8.     InitialWindowSize ← averageLength(E_F)
9.     for p in P do
10.        Let relevantScores be relevant scores for p to all entities in E_F
11.        for window in sliding_window(p, InitialWindowSize, step_size) do
12.            D, I ← index.search(Model.encode(window), len(E_F)) //搜索窗口相对于所有实体的距离
13.            relevantScores ← relevantScores+D[0].sorted
14.        end for
15.        topk_indexs ← np.argsort(relevantScores)[:k]
16.        topk_average_length ← calculateAverageLength(topk_indexs) //筛选最相近的 k 个实体计算新的窗口大小
17.        for window in sliding_window(p, topk_average_length, step_size) do
18.            D, I ← index.search(Model.encode(window), top_n) //使用新的窗口大小进行匹配
19.            e_F, distance ← getEntityAndDistance(D, I)
20.            if distance>threshold do
21.                Matches ← Matches+(p, window, e_F)
22.            end if
23.        end for
24.    end for
25.    return Matches
26. end function
```

获得文档中的功能实体匹配位置后, LLM-Extractor 从这些存在匹配的语句中筛选包含至少一个软件配置项

的语句. LLM-Extractor 将这些语句输入给 LLM 并提供少量样本案例, 抽取 E_C 对软件功能 E_F 的依赖关系, 最后将找到的 *Require* 关联关系加入 CFRG 中.

(2) E_C-E_C 关系抽取

这一步的目的是抽取配置项之间的直接关联关系. 由于第 3.1.1 节中的配置标注步骤已经标注了每个语句中的软件配置项, LLM-Extractor 只需筛选出直接包含多个配置项的语句, 进而引导 LLM 分析这些语句抽取多个配置之间的直接关联. 多个配置之间的直接关联可以分为 3 种情况: 1) *LeadTo* 关系, 表示一个或多个配置条件成立时, 会导致特定的影响, 该种影响可能是配置生效相关的, 也可能是系统功能状态相关的. 例如从文档语句“If fsync is set to off, the setting of wal_sync_method does not take effect”中可以提取得到: (fsync==off) *LeadTo AtStatus* (wal_sync_method, doesn’t take effect). 2) *Require* 关系, 该种关联关系结构与条件导致型相反, 由配置条件作为配置生效或者特定系统功能正常运行的必要条件. 例如从配置文档语句“resource_track_duration is valid only when enable_resource_track is set to on”中可以提取: Effect(resource_track_duration, valid) *Require* (enable_resource_track==on). 3) Otherwise 关系, 这种关系存在于配置建议、配置警告语句中, 句中建议或者要求配置项应满足一定的取值条件, 以达到系统正常运行等目的, 这些目的需要结合该句语句的上下文进行分析获得. 例如从 “You are advised to set max_cached_tuplebufs to a value greater than or equal to twice of max_changes_in_memory”中可以提取(max_cached_tuplebufs≥max_changes_in_memory) *Otherwise AtStatus*(cache for logical decoding not enough).

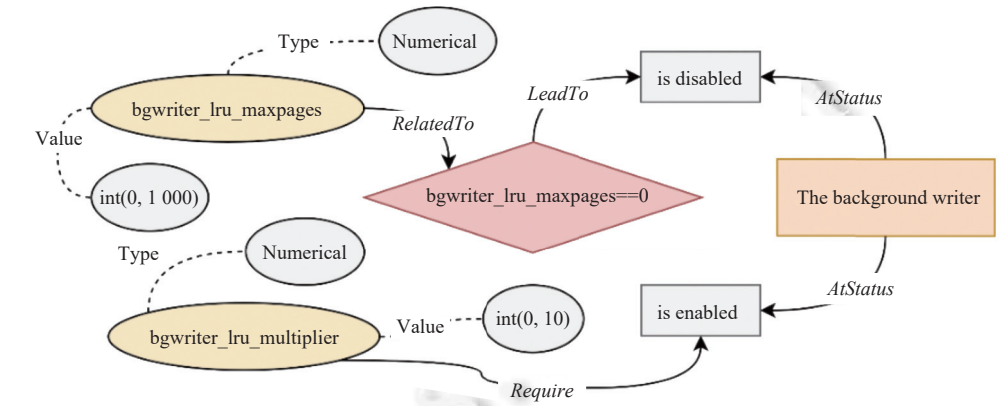
借助 LLM 提示工程, 辅以思维链和少样本学习方法, LLM-Extractor 分 3 个步骤提取上述的 3 种关联关系: 1) 判断是否存在可检测的配置条件; 2) 提取所有配置条件并使用合取析取连接; 3) 判断这些组合配置条件是否构成上述 3 种逻辑关系, 并输出关联关系. 获得输出后 LLM-Extractor 在 CFRG 中添加新的关联关系和配置状态实体. 对于在 3 种逻辑关联关系涉及的 Effect, LLM-Extractor 分成两类处理: 1) 对于配置生效的影响, LLM-Extractor 在 CFRG 中添加一个配置状态实体 E_{CS}, 并添加 *RelatedTo* 关系将 E_{CS} 关联到对应的软件配置项 E_C; 2) 不涉及软件配置项, 这种情况 Effect 可能可以关联 CFRG 中已有的软件功能实体 E_F, 即 Effect$\in E_{FS}$, 此时 LLM-Extractor 通过向量匹配的方式匹配 Effect 与 E_{FS}, 若相似度高于特定阈值 s, 则认为该 Effect 与该 E_{FS} 相同. 若 Effect 无法匹配到已有的 E_{FS}, LLM-Extractor 在 CFRG 中添加新的 E_{FS} 节点, 但是该节点不关联已有的软件功能实体 E_F. 对于 E_{Cond} *Otherwise* E_{FS} 关系, 可以把这种关系实际等同于 *not* E_{Cond} *LeadTo* E_{FS}, 并在 CFRG 中添加相应的条件节点 E_{Cond} 和 *LeadTo* 关系.

(3) E_F-E_F 关系抽取

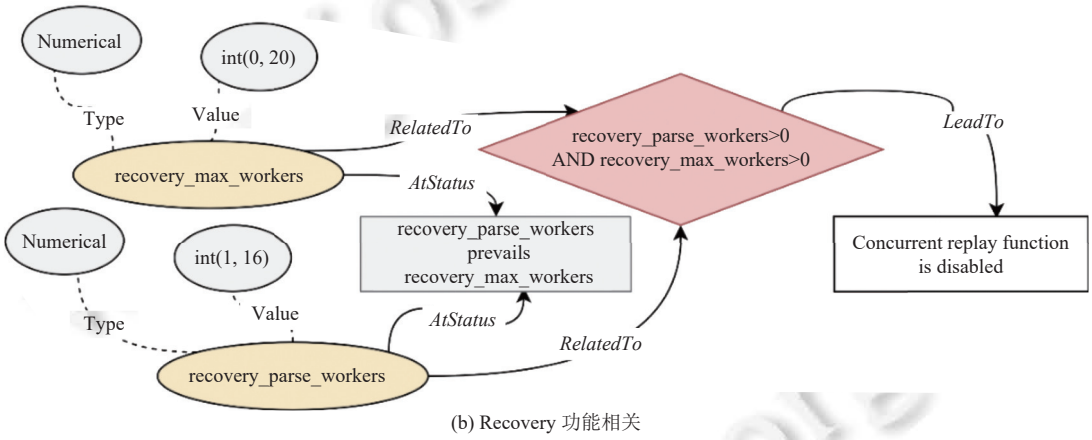
这一步的目的是在 CFRG 上进一步添加软件功能 E_F 之间存在的依赖关系, 进而链接可能通过多个 E_F 节点相关联的 E_C 节点. 通过第 2.2.1 节的经验研究我们发现, 软件功能实体之间也可能存在一定的约束关系, 例如在图 5 中, audit logs 依赖 auditing function 的开启. 同时通过图 5 的例子可以发现, 这些功能间依赖可能传递配置约束关系. 因此抽取功能间的依赖关系, 有助于提取经过多层传递形成的复杂配置间约束.

通过第 2 节的配置文档分析我们发现, 软件功能实体及其状态间的依赖关系具有 3 方面特点: 1) 一般不直接出现在配置文档的文本描述中, 需要研究人员结合专家经验判断; 2) 一般出现在同一章节涉及的功能实体之间. 例如在配置项 wal_receiver_status_interval 的描述中提到“If this parameter is set to 0, the standby server does not send information”, 根据 CFRG 初始化规则, 我们从该句中提取软件功能实体“the standby server”以及对应的功能状态“does not send information”, 同时补充相反状态“can send information”. 而在同一章节的配置项 hot_standby 的描述中, 我们发现 hot_standby 的功能是“hot_standby specifies whether to allow connections and queries on a standby server during its recovery”. 研究人员基于软件使用和运维经验, 认为“connections and queries on a standby server”应当依赖于“the standby server can send information”. 从上述的例子中我们发现, 研究人员的判断过程结合了涉及功能描述的文本以及自身经验, 而且这两个功能实体均出现在“standby server”章节中. 我们认为 LLM 在使用大量各类数据预训练的过程中可以获得此类经验, 因此 LLM-Extractor 调用 LLM 接口, 引导 LLM 判断不同文段描述中的功能实体是否存在依赖关系.

在 CFRG 的构建过程中, LLM-Extractor 利用 LLM 对非结构化文本的理解和分析能力, 从形式多样的配置文本描述中搜索并归纳存在的 E_C-E_F 关系、E_C-E_C 关系和 E_F-E_F 关系, 从而高效地将配置、功能之间复杂多样的关联关系以图结构的形式进行描述, 将不同的配置项通过直接或者间接的方式链接起来, 便于后续的多配置关联子图搜索. 完成了 E_C-E_F 关系抽取、E_C-E_C 关系抽取和 E_F-E_F 关系抽取后, CFRG 的构建就已经完成, 如图 9 展示了一个 CFRG 补全图的部分子图示例. 相比图 8 展示的 CFRG 初始化示例图, CFRG 补全图中具有更多的配置功能关联关系以及多配置关联关系. 在第 3.2 节中本文将介绍从 CFRG 中挖掘多配置关联子图并推断配置约束的过程.



(a) Background writer 功能相关



(b) Recovery 功能相关

图 9 CFRG 补全图示例

3.2 基于多配置关联子图的配置间约束推断

基于多配置关联子图的配置间约束推断是配置间约束提取的最终步骤, LLM-Extractor 基于构建完成的 CFRG, 搜索多配置关联子图, 把关联子图信息作为上下文输入给 LLM, 引导 LLM 推断潜在的配置间约束关系. 因此配置间约束推断可以分为多配置关联子图搜索和约束推断两个步骤, 本节将依次介绍这两个步骤.

3.2.1 多配置关联子图搜索

该步骤以配置条件节点为起点, 使用图算法在 CFRG 上搜索可能存在的多配置关联路径, 并记录这些关联路径, 最终将关联路径合成的关联子图输出. 多配置关联子图搜索的具体流程如算法 2 所示. 在算法 2 第 3 行, LLM-Extractor 以 CFRG 中的配置条件节点为搜索起点, 搜索与之存在 *RelatedTo* 以及 *Require* 关系的配置节点, 将这些信息加入关联路径中, 第 5–10 行展示了这一过程. 随后在算法 2 的第 13–19 行 LLM-Extractor 搜索了与配置条件节点存在 *LeadTo* 关系的功能实体, 并进一步搜索了该实体相关联的其他配置项, 并把这些信息加入关联路径表, 如果最终 l_{path} 中关联了超过一个配置项, 则根据 l_{path} 从 CFRG 中提取关联子图 G_{sub}, 并将 G_{sub} 返回. 在后续的配置

间约束推断步骤中, 多配置关联子图 G_{sub} 将作为重要的上下文依据.

算法 2. 多配置关联子图搜索算法.

输入: 输入配置-功能关联图 G;
输出: 多配置关联子图 G_{sub}.

```
1. function MutiConfRelationSearch(G)
2.    Let L_path be the return list
3.    for e_Cond in G.E_Cond do
4.       Let l_path be the relation path
5.       for e_C in G.E_C do //搜索相关配置
6.          if e_C RelatedTo e_Cond do
7.             l_path ← l_path + e_C RelatedTo e_Cond
8.          end if
9.          if e_C Require e_Cond do
10.            l_path ← l_path + e_C Require e_Cond
11.         end if
12.      end for
13.      for e_FS in G.E_FS do
14.         if e_Cond LeadTo e_FS do
15.            l_path ← l_path + e_Cond LeadTo e_FS
16.            e_F ← SearchFunctionEntity(e_FS) //找到关联的功能节点
17.            l_path ← l_path + SearchDenpendentPath(e_F) //搜索该功能节点的依赖关系
18.            for e_FS_Other in e_F.Status do
19.               if e_C Require e_FS_Other do
20.                  l_path ← l_path + e_C Require e_FS_Other
21.               end if
22.            end for
23.         end if
24.      end for
25.      if CountConf(l_path) >1 do
26.         L_path ← L_path + l_path
27.      end if
28.   end for
29.   G_sub ← GetSubGraph(G, L_path)
30.   return L_path
31. end function
```

3.2.2 配置间约束推断

完成多配置关联子图搜索之后, LLM-Extractor 将多配置关联子图作为上下文提供给 LLM, 引导 LLM 推断可能存在的配置间约束关系. 根据第 2.2.1 节经验研究的相关发现, 我们把软件配置约束分为如下 3 种类型.

● 配置生效约束. 配置生效约束指一个配置项的生效会受到其他配置项的影响, 这种影响关系可能通过软件功能实体的状态传递, 即 E_C-E_F-E_C, 也可能直接体现为软件配置项的直接关联, 即 E_C-E_C. 例如在图 9 上半部分的

子图中, 配置项 bgwriter_lru_multiplier 生效要求 the background writer 的状态是 enabled, 而 bgwriter_lru_maxpages 值为 0 会导致 the background writer 的状态是 disabled, 从该关联子图中我们可以经由 the background writer 功能状态的传递关系, 推断出配置间约束:

(bgwriter_lru_maxpages==0) *LeadTo* NotEffect(bgwriter_lru_multiplier).

该过程与图 3 展示的人工思维链类似, 通过 E_C-E_F-E_C 关系推断配置间约束. 而在图 9 下半部分的子图中, recovery_max_workers 和 recovery_parse_workers 均大于 0 会导致“recovery_parse_workers prevails recovery_max_workers”, 从而可以推断出配置间约束:

(recovery_parse_workers>0 AND recovery_max_workers>0) *LeadTo* NotEffect(recovery_max_workers).

该过程即直接通过 E_C-E_C 的多配置关联关系推断配置间约束.

● 配置共同行为约束. 配置共同行为约束指多个配置项相关的配置条件同时满足时会导致某项系统行为发生. 例如在图 9 下半部分的子图中, recovery_max_workers 和 recovery_parse_workers 均大于 0 会导致 concurrent replay function is disabled, 这说明这两个配置组成的条件共同决定了该系统行为, 构成了共同行为约束.

● 配置建议约束. 配置建议约束指软件配置被建议设置为特定值或者满足特定条件, 如果该条件不满足, 系统可能产生预期之外的后果. 例如配置项 maintenance_work_mem 和 work_mem 所关联的配置条件 maintenance_work_mem≥work_mem 不满足时, 会导致“system might not function optimally during automatic vacuuming or maintenance operations”, 这体现了配置建议约束, 即 maintenance_work_mem 被建议设置为大于 work_mem 的值.

一条配置间约束关系由一组明确的配置条件和相应的提示信息组成. 因此在配置间约束推断阶段, LLM-Extractor 读入一组多配置关联子图信息, 并加入相关配置的类型、范围等必要上下文信息, 引导 LLM 在关联路径中推断是否存在上述的 3 种配置间约束, 并输出条件和对应的提示信息. 该步骤的提示模板如图 10 所示.

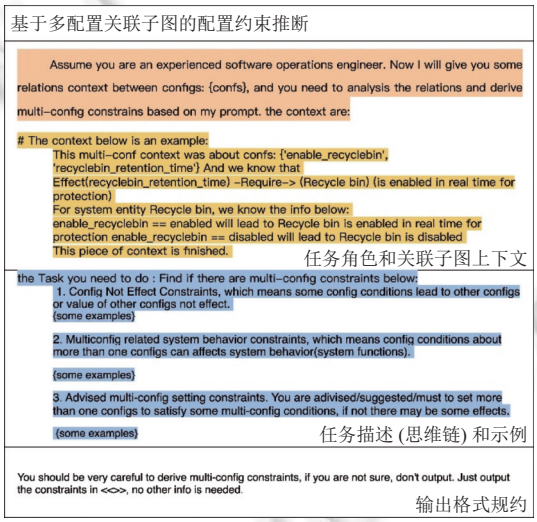


图 10 配置间约束推断提示词结构

如图 10 橙色部分介绍了配置间约束推断任务中 LLM 的任务角色, 黄色部分将关联子图中的信息以自然语言的方式写入提示词中, 便于模型理解和分析, 蓝色部分则说明了 3 种类型的配置间约束, 并给出少量示例, 提升模型的分析推断效果, 提示词最后约定输出格式, 便于识别返回结果.

4 实验和评估

为了验证本文提出方法的有效性, 我们使用 Python 实现了 LLM-Extractor, 集成了多个广泛使用的 LLM 的调用接口, 并使用来自多个大型开源软件系统的配置文档数据进行实验和评估, 本节主要围绕以下 3 个问题进行实

验并开展验证.

- RQ1: LLM-Extractor 相较已有基于自然语言分析的方法效果如何?
- RQ2: 本文使用 LLM 构建 CFRG 的效果如何?
- RQ3: LLM-Extractor 相较已有的基于程序分析的方法各有何优劣?

4.1 实验设置

实验设置分为数据收集和模型选择两个部分, 本节将分别介绍.

4.1.1 实验数据收集

根据已有工作的研究经验, 我们选择了 openGauss 5.0.0[3]、PostgreSQL 12[49]、Squid 6[1]这 3 个被广泛使用的开源软件系统作为实验研究对象, 选择这 3 个系统的原因是这 3 种软件可调整的运行时配置众多, 且已经被广泛研究, 除此之外, 这 3 种软件提供了明确的配置-配置描述对应关系. 本文使用这些系统的官方文档作为数据源, 构建数据集. 文本数据采集和预处理的方法在第 3.1.1 节中已经详细阐述, 此处不再赘述. 完成数据采集和预处理后, 我们根据收集到的配置文本数据, 通过 3 名研究人员人工阅读文本综合分析的方式, 尽可能找出文本中存在的配置间约束, 作为后续验证方法效果的依据. 人工分析的流程分为 3 个步骤: (1) 由两名研究人员分别阅读每个软件的配置文档, 首先分析包含多个配置项的描述语段, 提取其中可能存在的配置间约束. (2) 两名研究人员再次阅读每个配置的描述, 重点分析同一目录下的配置项, 分析是否存在跨文段产生关联的配置间约束. (3) 第 3 名研究人员将前两阶段分析获得的配置间约束, 进行去重并整合, 如果出现不一致的情况, 由 3 名研究人员共同商议有争议的约束规则. 3 名研究人员使用了两周时间来完成这项工作, 最终共从 3 个软件系统的配置文档中提取出 306 条配置间约束, 构成了本文进行实验评估的数据集. 数据集具体情况如表 4 所示.

表 4 数据集收集情况

软件名称	版本	软件配置个数	配置文本语句总数	配置间约束数量
openGauss	5.0.0	768	7952	193
PostgreSQL	12	319	2568	86
Squid	6	330	4110	27

4.1.2 模型选择

根据已有 LLM 相关工具的研究经验, 我们综合考虑了模型请求开销、响应时间、模型能力等因素, 选取了 openAI 的 GPT-4o-mini[51]和 GPT-4o[52]进行进一步的实验研究. GPT-4o 和 GPT-4o-mini 是基于 OpenAI GPT-4 技术的衍生模型, 其中 GPT-4o 是一个基于 GPT-4 的优化版本, 旨在保持 GPT-4 的核心能力, 同时在效率上进行了提升, 该模型较原始的 GPT-4 模型更轻量化, 能够在特定场景下提供类似的语言理解和生成效果; 而 GPT-4o-mini 是 GPT-4o 的更小型版本, 同时接口调用价格也大幅度低于 GPT-4o 和 GPT-4, 由于模型规模更小, GPT-4o-mini 在逻辑推断和语言生成能力上不及 GPT-4o 和完整的 GPT-4, 但仍保留了一定的语言生成和理解能力.

4.1.3 指标选择

考察精确率 (*Precision*)、召回率 (*Recall*)、准确率 (*Accuracy*) 和 *F*1 *score* 这 4 个指标, 其定义如公式 (1)–公式 (4) 所示, 其中 $\boldsymbol{S}$ 和 $\boldsymbol{G}$ 分别表示实验结果中正确约束关系或者关联关系集合和事先确定的标准答案集合, $\boldsymbol{N}$ 表示不存在关联关系的提示结果集合.

$$Precision = \frac{|\boldsymbol{S} \cap \boldsymbol{G}|}{|\boldsymbol{S}|} \tag{1}$$

$$Recall = \frac{|\boldsymbol{S} \cap \boldsymbol{G}|}{|\boldsymbol{G}|} \tag{2}$$

$$Accuracy = \frac{|\boldsymbol{S} \cap \boldsymbol{G} \cup \boldsymbol{N}|}{|\boldsymbol{S} \cup \boldsymbol{G} \cup \boldsymbol{N}|} \tag{3}$$

$$F1\ score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4}$$

4.2 LLM-Extractor 与已有基于自然语言分析的方法对比评估 (RQ1)

本节我们将评估对比 LLM-Extractor 和已有配置约束提取方法 PracExtractor 的效果, PracExtractor 是一种基于自然语言分析的配置约束提取方法, 它通过潜在约束语句识别和约束模板匹配两个步骤提取配置约束, 直接输出对应的配置条件和影响, 在多种不同的系统上有广泛的实验应用[26]. 除此之外还通过消融实验验证多配置关联子图作为上下文对配置间约束提取任务的作用.

4.2.1 和已有方法的对比

在 CFRG 初始化完成之后, LLM-Extractor 在 CFRG 上搜索多配置关联子图并使用这些关联子图信息推断配置间约束. 我们使用 Python 3.9 实现了多配置关联路径搜索方法, 基于第 4.2 节完成关联关系补全后的 CFRG (使用 GPT-4o 模型), 搜索多配置相关的关联子图, 并基于关联路径推断配置间约束. 我们选择了已有工作 PracExtractor 作为对比方法, 以判断 LLM-Extractor 在配置间约束提取任务上的效果. 为了评估不同方法在配置间约束提取上的效果, 我们把事先分析提取出的配置间约束记为全集 $\boldsymbol{G}$. 使用不同方法获得的配置间约束集合记为 $\boldsymbol{S}$, 我们使用精确率 *Precision*、召回率 *Recall*、*F*1 *score* 来评估方法的效果. 实验结果如表 5 所示. 从表 5 中我们可以看出, LLM-Extractor 在精确率和召回率上的表现较为均衡, 在 3 种系统实验上的配置间约束提取精确率在 0.76–0.81 之间, 召回率在 0.74–0.81 之间, *F*1 分数均超过了 0.75. PracExtractor 的精确率较高, 在 0.83–0.91 之间, 经过分析我们认为这是由于 PracExtractor 基于正则表达式的模式匹配, 这种方法的优势是严格符合预定义模式的约束关系才会被匹配, 从而提高了精确率; 而 PracExtractor 的召回率远低于 LLM-Extractor, 只有 0.21–0.37, 这是由于严格的模式匹配没有充分考虑文档的语义信息, 导致包含配置约束但是与模板不符的语句被忽略, 除此之外, LLM-Extractor 可以综合多个不同配置的描述文本, 并分析配置之间通过软件功能状态传递的关联关系, 从而得出配置间约束关系, 而 PracExtractor 只能从单个语句中匹配约束关系, 这导致 PracExtractor 无法提取这些通过软件功能状态传递的约束关系.

表 5 PracExtractor 与 LLM-Extractor 提取配置间约束效果比较

软件名称	$\boldsymbol{G}$ count	方法	$\boldsymbol{S}\cap\boldsymbol{G}$ count	*Precision*	*Recall*	*F*1 *score*
openGauss	193	LLM-Extractor (GPT-4o)	156	0.808 3	0.804 1	0.806 2
		PracExtractor	42	0.875 0	0.217 6	0.348 5
PostgreSQL	86	LLM-Extractor (GPT-4o)	65	0.764 7	0.755 8	0.760 2
		PracExtractor	20	0.833 3	0.232 6	0.363 6
Squid	27	LLM-Extractor (GPT-4o)	20	0.769 2	0.740 7	0.754 7
		PracExtractor	10	0.909 1	0.370 4	0.526 3

4.2.2 消融实验

为了验证多配置关联子图作为提示上下文的作用, 我们进行了消融实验, 在消融实验中直接依次将每个配置的文本描述装填入提示模板中, 而不提供多配置关联子图作为上下文, 直接请求 GPT-4o 接口得到实验结果, 我们把这种基线方法记为 Direct Prompt. 实验结果以及和 LLM-Extractor 的对比如表 6 所示. 从表 6 中我们可以发现, 相比于基线方法, LLM-Extractor 在精确率上有不同程度的提高, 其中在 openGauss 实验中提升幅度最小, 在 Squid 实验中的提升幅度最大, 经过分析, 我们发现这是由于 LLM 在分析 Squid 的过程中出现了更多主观臆断. 我们认为这是因为 openGauss 配置文档撰写方式较为规范, 而 Squid 配置文档常出现缺失主语宾语、指代不明的情况, 语料质量相对较低, 容易导致 LLM 猜测文本意图的情况. 而在召回率上, LLM-Extractor 均明显高于基线方法, 我们认为这主要是由于 LLM-Extractor 能够分析通过软件功能状态传递的配置约束关系, 这需要综合多个不同配置项的描述文本, 而基线方法只具有分析单个配置文本的能力.

4.3 使用 LLM 构建 CFRG 的效果评估 (RQ2)

CFRG 是后续 LLM-Extractor 进行配置约束推断的数据基础, 因此 CFRG 的构建质量对约束提取效果具有重要影响. CFRG 的构建分为初始化、配置-功能实体关联、多配置关联等步骤, 本节将依次进行评估.

表 6 Direct Prompt 与 LLM-Extractor 提取配置间约束效果比较

软件名称	G count	方法	$S \cap G$ count	*Precision*	*Recall*	*F*1 *score*
openGauss	193	LLM-Extractor (GPT-4o)	156	0.8083	0.8041	0.8062
		Direct Prompt (GPT-4o)	66	0.6535	0.3420	0.4490
PostgreSQL	86	LLM-Extractor (GPT-4o)	65	0.7647	0.7558	0.7602
		Direct Prompt (GPT-4o)	24	0.5455	0.2791	0.3692
Squid	27	LLM-Extractor (GPT-4o)	20	0.7692	0.7407	0.7547
		Direct Prompt (GPT-4o)	12	0.22	0.4074	0.2857

4.3.1 CFRG 初始化评估

LLM-Extractor 在 CFRG 初始化阶段读取每个配置项对应的软件配置文档, 并提取配置项的类型以及合法取值, 即从文本中识别当前配置项 E_C 的类型属性 Type, 以及合法取值属性 Value. 除此之外, 根据该配置项的类型, LLM-Extractor 进一步提取该配置项取值条件对特定系统功能的边际影响, 将这些系统功能以及对应的状态抽取为 E_F 和 E_{FS} 节点, 将导致该影响的配置条件抽取为 E_{Cond} 节点, 并添加 *RelatedTo* 关系与 E_C 关联. 由于该任务请求量巨大且推理的复杂程度较低, 我们调用轻量化的 GPT-4o-mini 接口完成了 CFRG 的初始化实验, 并人工分析了实体抽取的正确性, 该过程由两名研究人员分别完成, 再由第 3 名研究人员整合判断结果, 如有分歧则共同讨论得出结论. CFRG 初始化的实验结果如表 7 和表 8 所示. 表 7 展示了使用 LLM 提示工程抽取实体的次数情况, 表 8 展示了每种实体抽取的精确度. LLM-Extractor 针对每个配置项的描述文本生成提示词并调用 LLM 接口, 由于每个配置项都具有类型和取值属性, 因此 Type 和 Value 的提取次数和配置个数相同. 从表 8 中可以发现, Type 和 Value 抽取的精确率较高, 在 openGauss 和 PostgreSQL 的实验中均超过了 0.97, 这是由于软件配置设计时就遵循先进行类型设计和合法值约定的规则, 因而文档中一般包含配置类型和取值范围的描述. Squid 实验中, Type 和 Value 抽取的精确率略低在 0.93 左右, 这是由于 Squid 的文档撰写相对更加简略. 在 E_F 和 E_{FS} 实体抽取任务上, 由于涉及配置功能理解和边界条件抽取, 具有更高的难度, 在这个任务上 GPT-4o-mini 的表现稍差, 准确率在 0.85–0.93 之间. 由于 LLM-Extractor 可能从不同语料中抽取出相同 E_F 和 E_{FS} 实体, E_F 和 E_{FS} 实体需要进一步去重和消歧. 除此之外, 还要添加与 E_{FS} 实体相反的状态实体, 便于后续的逻辑推断, 具体方法如第 3.1.2 节中相反状态添加所示. 完成去重和消歧和相反状态添加之后的 E_F 和 E_{FS} 实体数量如表 9 所示. 实体数量相较于抽取次数均有所减少, 而由于相反状态实体的添加, E_{FS} 实体数量相较于抽取次数下降幅度较小或者有略微增加. 实验表明, GPT-4o-mini 在文本理解和分类、关键范围抽取问题上具有良好的效果, 在逻辑关系抽取上的表现则稍弱.

表 7 针对 GPT-4o-mini 模型的抽取次数

软件名称	配置个数	Type count	Value count	E_F count	E_{FS} count
openGauss	768	768	768	529	915
PostgreSQL	319	319	319	261	395
Squid	330	330	330	206	319

表 8 针对 GPT-4o-mini 模型抽取准确率

软件名称	配置个数	Type *Precision*	Value *Precision*	E_F *Precision*	E_{FS} *Precision*
openGauss	768	0.9883	0.9843	0.8809	0.9180
PostgreSQL	319	0.9906	0.9781	0.8697	0.9063
Squid	330	0.9394	0.9363	0.8981	0.9310

表 9 E_F 和 E_{FS} 实体去重消歧结果

软件名称	配置个数	E_F count	E_{FS} count
openGauss	768	396	803
PostgreSQL	319	217	394
Squid	330	163	331

4.3.2 配置-功能实体关联抽取实验评估

该阶段 LLM-Extractor 读取每个配置项的配置文档, 在文档中使用滑动窗口算法识别文段中的软件功能实体 E_F, 我们使用 Python 3.9 实现了滑动窗口算法, 并使用 bge-large-en-v1.5 模型计算文本窗口的向量表示. 完成实体识别后, LLM-Extractor 根据匹配结果筛选出可能关联其他配置项的匹配语句 (即排除仅与当前配置项关联的功能实体), 随后将这些语句和对应的配置文本语段装填入提示模板中, 接着调用 GPT-4o-mini 和 GPT-4o 接口分别进行了配置-功能实体关联抽取实验. 我们首先人工分析了每条提示, 由两名研究人员, 分别独立地分析提示词并得出关联关系, 若两人结论不同则交由第 3 名研究人员判断, 通过共同讨论确定标准答案. 在实验运行完毕后, 我们根据事先确定的标准答案对抽取的关联关系进行了人工评估和标注.

实验结果评估如表 10 所示. 从表中可以发现, GPT-4o 效果较好而 GPT-4o-mini 的效果欠佳, 这是由于配置-功能关系抽取涉及更复杂的文本理解和逻辑推断, 能力更强的 GPT-4o 相对于轻量化的 GPT-4o-mini 模型取得了更好的效果, 在精确率上有 14.2%–52.8% 的提升, 在召回率上则提升不大, 说明 LLM 倾向于将潜在的配置-功能关联均识别为关联关系. 在 openGauss 实验进行的 219 次提示中, GPT-4o 取得了超过 0.8 的精确率和超过 0.85 的召回率, $F1$ 分数超过了 0.83; 而在 PostgreSQL 实验的 167 次提示中, 使用 GPT-4o 的精确率略低, 为 0.725 5, 通过分析实验结果, 我们初步推断这可能是由于 openGauss 配置文档拥有比 PostgreSQL 配置文档更详细的配置功能描述, 从而降低了 LLM 理解文本的难度; 在 Squid 实验的 56 次提示中, GPT-4o 取得的召回率反而比 GPT-4o-mini 低 3.85%, 我们认为这是由于能力更强的 GPT-4o 拥有更为谨慎保守的策略, 而 Squid 实验相对较少的样本数可能会放大这种影响.

表 10 E_F-E_C 关联抽取实验结果

软件名称	提示次数	模型	*Precision*	*Recall*	*Accuracy*	*F1 score*
openGauss	219	GPT-4o-mini	0.597 4	0.859 8	0.670 9	0.705 0
		GPT-4o	0.817 4	0.862 4	0.846 1	0.839 3
PostgreSQL	167	GPT-4o-mini	0.634 9	0.909 1	0.796 9	0.747 7
		GPT-4o	0.725 5	0.880 9	0.857 1	0.795 7
Squid	56	GPT-4o-mini	0.545 5	0.8	0.767 9	0.648 6
		GPT-4o	0.857 1	0.923 1	0.946 4	0.888 9

4.3.3 多配置关联抽取实验评估

该阶段 LLM-Extractor 再次读取每个配置项的配置文档, 从中筛选出包含多个配置项的语句, 分别调用 GPT-4o-mini 和 GPT-4o 模型接口, 提取潜在的多配置逻辑关联关系. 和第 4.2.2 节配置-功能实体关联抽取部分类似, 我们事先人工分析了每条提示并通过交叉确认讨论出标准答案, 完成实验后再由研究人员对实验结果进行评估和标注. 实验结果如表 11 所示. 在使用 GPT-4o 的实验中, 方法都取得了更好的效果, 相比 GPT-4o-mini, 在精确度上有 26.25%–52.94% 的提升, 在召回率上则效果区别不大. 分软件来看, LLM-Extractor 在 openGauss 和 PostgreSQL 上取得了比在 Squid 上更好的效果, 前两者精确度均在 0.86 以上, 而后者精确度不足 0.77, 经过实验结果分析我们认为, 这可能是由于 Squid 的配置文档中缺少明确的配置条件表达, 需要 LLM 从复杂的自然语言描述中提取出明确的配置条件, 这对模型的能力提出了更高的要求. 同时我们还注意到, 相比 openGauss 和 PostgreSQL 实验, 在 Squid 实验中使用 GPT-4o-mini 时具有更严重的效果下降, 分析实验结果我们发现, 这可能是由于 GPT-4o-mini 相对于 GPT-4o 具有“更不谨慎”的特点, 更容易把形式各异的自然语言表述均识别为配置条件, 从而影响了关联抽取的精确率.

4.3.4 功能实体间关联抽取实验评估

由于 Squid 配置文档并没有按照功能模块进行划分, 我们在 openGauss 和 Squid 的部分主要功能章节 (涉及 standby server、background writer、auditing、genetic query) 中进行了实验, 尝试抽取功能实体间的依赖关系. 对于 openGauss 系统, 这些章节涉及 15 个功能实体节点, 我们以分组为单位, 在分组内取 2 的组合, 共生成 37 组功能节点对, 从而生成 37 条提示词. 在 PostgreSQL 中以同样的方式, 生成了 45 条提示词. 为了评估 LLM 输出效果,

我们事先人工分析了每条提示词并交叉确认答案, 完成实验后再由研究人员对实验结果进行双重评估和标注, 最后整合评估结果. 实验结果如表 12 所示, 从表中我们可以发现, LLM 在此类需要结合知识经验进行推理的问题上表现相对较差, 使用 GPT-4o 的实验精确率在 0.68–0.72 之间, 而使用 GPT-4o-mini 的实验精确率仅在 0.25–0.32 之间, 分析实验数据我们认为 GPT-4o-mini 更倾向于认为两个功能节点之间存在依赖关系, 从而输出了更多的错误依赖. 综合召回率等其他指标我们认为, LLM 在此类综合理解推理任务上的应用能力仍有待加强.

表 11 E_C-E_C 关联抽取实验结果

软件名称	提示次数	模型	*Precision*	*Recall*	*Accuracy*	*F*1 *score*
openGauss	114	GPT-4o-mini	0.6966	0.9538	0.7368	0.8052
		GPT-4o	0.8795	0.9481	0.9125	0.8772
PostgreSQL	45	GPT-4o-mini	0.6765	0.9583	0.7333	0.7931
		GPT-4o	0.8667	0.9630	0.8889	0.9123
Squid	37	GPT-4o-mini	0.5	0.8462	0.6486	0.6286
		GPT-4o	0.7647	0.9286	0.8649	0.8387

表 12 E_F-E_F 关联抽取实验结果

软件名称	提示次数	模型	*Precision*	*Recall*	*Accuracy*	*F*1 *score*
openGauss	37	GPT-4o-mini	0.2581	0.8	0.3243	0.3902
		GPT-4o	0.7143	0.9091	0.8649	0.8
PostgreSQL	45	GPT-4o-mini	0.3182	0.7	0.6	0.4375
		GPT-4o	0.6842	0.8667	0.8222	0.7647

4.4 LLM-Extractor 与已有程序分析方法的对比评估

本节将通过实验探究 LLM-Extractor 与已有程序分析方法相比各有何优劣.

4.4.1 对比方法和实验系统选择

本节选择了 cDep 方法[13]作为对比方法, cDep 是一种基于 Soot[53]程序分析框架开发的配置间约束提取方法, 该方法可以分析使用 Java 和 Scala 语言编写的软件源代码, 通过控制流和数据流分析提取源代码中存在的软件配置间约束. cDep 不仅可以提取单个软件内的配置间约束, 还能分析同软件栈内跨不同软件的配置间约束, 本节仅探究单个软件中配置间约束的提取任务.

为了对比 LLM-Extractor 和 cDep 方法相比各有何优劣, 本节选择 HDFS[54]作为实验系统, HDFS 是 Hadoop 生态中的重要组件, 它使用 Java 编程语言编写, 并且具有完整的具有配置对应关系的配置文档, HDFS 系统的基本情况如后文表 13 所示.

4.4.2 LLM-Extractor 在 HDFS 上的提取效果

本节使用 LLM-Extractor 分析了 HDFS 配置文档[55]中出现的 425 个配置项, 根据文档描述构建 HDFS 文档的 CFRG, 并基于该 CFRG 分析提取配置间约束, 同时本文为了比较 LLM-Extractor 与 cDep 在配置间约束提取任务上的各自优劣, 还分析了 cDep 提取 HDFS 中配置依赖的实验数据, 如表 14 所示. 从实验结果可以看出, LLM-Extractor 在 HDFS 系统上也表现出良好的泛化能力, 提取出 37 条配置间约束, 精确率超过 0.82, 其中绝大部分 (33/37) 为生效类型约束. cDep 在 Hadoop 软件栈中提取配置间约束的精确率为 87.9%[13], 在这一项上整体优于 LLM-Extractor, 本文认为这得益于 cDep 较为先进的程序分析方法设计, 以及大语言模型相对不稳定的输出. 具体在 HDFS 系统上, cDep 共提取出 93 条软件内配置依赖.

通过对二者输出结果的分析本文发现两者所提取的配置约束维度不同, 例如 cDep 输出结果中“Control Dependency, dfs.balancer.keytab.enabled, dfs.balancer.keytab.file, org.apache.hadoop.hdfs…Balancer, *”表示了一条控制依赖, 但是缺乏关键的控制条件和违反后果, 需要结合进一步的人工分析和测试确认. 而 LLM-Extractor 的输出结果“dfs.datanode.peer.stats.enabled==false *LeadTo* NotEffect(dfs.datanode.outliers.report.interval)”中则包含了明确

的配置条件和条件影响. 鉴于两者在方法适用场景和输出内容上的差异, 接下来本文将进一步从多个方面对两者进行对比分析.

表 13 HDFS 基本情况

内容	信息
软件名称	HDFS
版本	2.9.2
软件配置个数	425
配置文本语句总数	1 075
代码行数	644k

表 14 HDFS 实验情况比较

方法	工具输出结果	数量
LLM-Extractor (GPT-4o)	提取配置间约束数量	37
	生效约束	33
	行为约束	2
	建议约束	2
cDep	提取配置依赖数量	93
	控制依赖	51
	行为依赖	20
	取值依赖	11
	重写依赖	2
	默认值依赖	9

4.4.3 LLM-Extractor 与 cDep 的对比分析

从方法适用的场景来看, 本文发现 cDep 与 LLM-Extractor 有明显区别. cDep 适用于能够获取完整源代码的开源软件, 而 LLM-Extractor 适用于各种开源软件或闭源商业软件. 而从工具输出的提取结果的形式来看, 二者也有显著的不同, cDep 根据配置的代码逻辑进行依赖分类, 该分类以特定的程序模板作为依据, 体现了配置变量影响程序控制流、数据流的情况. 从 cDep 论文和实验结果中可以看出, cDep 所提取的配置约束主要反映程序逻辑, 若要进一步探索违反约束关系可能导致的后果, 需要进一步测试, 这部分在 cDep 原文献中也有提及. 且 cDep 自动化工具只能输出约束出现的函数、类以及涉及的配置, 例如提取结果“Control Dependency, dfs.balancer.keytab.enabled, dfs.balancer.keytab.file, org.apache.hadoop.hdfs…Balancer, *”中指明了 dfs.balancer.keytab.enabled 和 dfs.balancer.keytab.file 之间存在控制依赖, 但是进一步判断依赖条件以及违反依赖的后果, 需要进一步人工分析和测试验证. 因此本文认为 cDep 所提取的约束属于运维流程中的中间数据, 无法直接自动化获得约束条件以及违反后果. LLM-Extractor 所提取的约束是整合多段配置文档综合分析的产物, 因此 LLM-Extractor 可以直接提取配置约束的明确条件并输出违反该条件的后果, 例如“dfs.datanode.peer.stats.enabled==false *LeadTo* NotEffect(dfs.datanode.outliers.report.interval)”中包含明确的配置条件和后果, 因此 LLM-Extractor 所提取的约束是宏观的、可静态验证的.

通过上述的输出内容分析, 本文认为两者所提取的配置约束维度不同, 无法在数量上直接对比分析. 因为 cDep 聚焦于提取代码中的约束模式, 即微观层面约束, 例如, if(editsDirs.isEmpty()){… return getStorageDirs(conf, “dfs.namenode.name.dir”)} //(editsDirs store the value of dfs.namenode.edits.dir), cDep 则认为 dfs.namenode.edits.dir 和 dfs.namenode.name.dir 之间存在一条控制依赖, 但是由于配置在源代码中会多次出现, 该代码片段无法说明这 dfs.namenode.edits.dir 和 dfs.namenode.name.dir 在源代码的所有位置都满足控制关系, 因此无法断言在整个软件运行的层面上, dfs.namenode.name.dir 被 dfs.namenode.edits.dir 控制. 而 LLM-Extractor 所提取的配置约束其来源是开发人员撰写的软件文档, 其中记录的配置影响具有宏观意义. 因此本文认为两者提取配置约束的维度不同. cDep 侧重在尽可能全面地挖掘源码中配置项之间的依赖关联, 而 LLM-Extractor 侧重在尽快提供运维人员可直接检查的约束规则以及违反后果.

为了能在同一维度对比两种方法, 本文采用宏观到微观映射的方式, 逐条分析 LLM-Extractor 所提取的配置间约束, 并在 cDep 的提取结果中搜索是否存在对应的程序模式, 若不存在, 则分析不存在的原因. 通过对比分析, 本文发现在 LLM-Extractor 所提取的 37 条配置间约束中, 有 11 条 (29.7%) 可以在 cDep 的实验结果中找到相关配置依赖, 分析发现, 这类约束在源代码中具有清晰的代码模式. 而有 26 条 (70.3%) 配置间约束未能在 cDep 的实验结果中找到对应配置依赖, 通过分析本文发现主要有 3 种情况: 1) 配置间约束通过功能依赖传递, 而这种复杂的依赖关系难以通过控制流和数据流分析技术挖掘, 例如 dfs.block.local-path-access.user 配置依赖于功能“short-circuit local read”, 而功能“short-circuit local read”的开闭由配置项 dfs.client.read.shortcircuit 控制, 而通过人工细致阅读代

码 (HDFS release-2.9.2), 本文并未在源代码中找到 dfs.client.read.shortcircuit 直接控制 dfs.block.local-path-access.user 的程序链路; 2) 配置间约束能够在特定程序位置找到线索, 但是未能被 cDep 预先定义的程序模式匹配, 例如配置 dfs.namenode.servicerpc-address 在未设置时系统会默认使用另一个配置 dfs.namenode.rpc-address, 我们通过阅读代码研究了该条配置间约束在代码中的存在形式, 发现了如图 11 的代码形式, 从图 11 中可以看出, 两个配置经过了两次别名变换, 最后通过 getAddressesForNsIds 函数中的复杂循环逻辑完成默认值处理, 这种依赖形式未能被 cDep 的预定义模式覆盖, 因此未能被 cDep 提取; 3) 配置间约束提到的配置未出现在源代码中, 例如配置 dfs.http.client.failover.max.attempts 依赖于功能“WebHDFS client”, 而配置 dfs.webhdfs.enabled 控制功能“WebHDFS client”的开闭, 但是我们在源代码文件中仅在 hdfs-default.xml 和 WebHDFS.md 文件中找到相关关键字, 并未在 Java 源文件中找到相关变量引用. 通过对比我们发现, LLM-Extractor 可以综合分析多段文档, 从而找出部分难以从源代码中获取线索的配置约束关系.

图 11 HDFS 配置约束代码形式示例

从方法泛化能力和适配性来看, 两种方法均有一定的泛化能力. cDep 提出的程序模式挖掘方法理论上适配大部分编程语言, 所定义的控制约束、行为约束等约束模式在不同软件中也可以找到对应模式实例, 经过对目标软件源码的阅读分析再通过开发对应软件的程序分析工具, 可以完成迁移应用. 但是此过程面临多种挑战, 首先 Java、C++等不同语言的语言特性差别巨大, 需要大量前期调研, 并使用 Soot、LLVM 等不同的工具链开发程序分析工具, 除此之外, 即使是相同语言编写的软件, 编译器版本差异、代码风格不同也会给程序分析工具开发带来困难, 例如适配不同的 LLVM 版本, 又例如需要处理代码中的特殊情况以保证程序分析工具的正常运行. 因此 cDep 在不同软件之间迁移应用需要消耗较多人力和时间, 且无法分析缺乏源代码的商业软件. 而 LLM-Extractor 作为一种轻量化的工具, 可以适用于不同的软件, 尤其对于无法获得源代码的商业软件, 可以通过分析配置文档迅速提取配置约束. 因此本文认为 LLM-Extractor 具有更好的泛化能力.

通过多个方面的对比, 本文认为 LLM-Extractor 与 cDep 各有优劣, 有不同的适用场景和优势. cDep 侧重于尽可能全面地挖掘配置之间可能存在的依赖关系, 并通过人工确认和测试的方法来获得违反依赖导致的系统行为, 有利于更全面地分析配置约束, 但是需要能够获得和编译源代码, 且部分配置约束在源代码中难以挖掘或者没有显式的表现形式, 难以通过程序分析方法获取. LLM-Extractor 则侧重轻量化地分析各种软件, 只需获得软件的配置文档, 泛化能力较强. 虽然受制于配置文档的编写质量, LLM-Extractor 在分析全面性上有一定劣势, 但是可以快速获得包含确定性条件和条件影响的配置约束, 适用场景更加灵活.

4.5 有效性分析

本节主要分析 LLM-Extractor 以及评估实验可能存在的一些不足.

4.5.1 数据集有效性

由于目前并没有从软件文档中提取配置间约束的公开大规模数据集, 本文所使用的数据集均由参与本文的研究人员自行构建. 由于研究人员人力有限, 本文重点研究了配置约束集中出现的配置文档, 暂未考虑可能涉及配置的其他功能文档. 同时, 从文档中提取配置间约束构建标准答案集合需要耗费大量人力, 并且依赖不同软件领域的专家知识, 因此人工构建数据集的过程中难免会引入主观性, 可能引入部分错误, 对实验评估产生消极影响.

此外, 本文选择 PracExtractor 作为对比方法, 按照 PracExtractor 论文描述复现了该方法并进行测试评估, 由于涉及关键词认定和约束模板设计, 该复现方法可能与原方法有所差别, 这可能会影响实验评估.

4.5.2 方法的有效性

本文所提出的方法依赖 LLM 的能力, LLM 尽管在多个领域展现了强大的能力, 它仍然存在一些问题.

首先, LLM 依赖于大量的计算资源, 而能力越强的模型, 调用的成本越高. 例如 GPT-4o 的调用成本是 GPT-4o-mini 的 16.7 倍, 而 GPT-4 的调用成本是 GPT-4o 的 24 倍. 如果需要使用 LLM 全量分析更大规模的软件文档, 模型的调用费用对于研究人员是一笔不小的开销. 我们未来将继续研究如何在分析的不同阶段使用不同能力的模型, 来适配不同难度的任务, 同时在构建提示时使用更多的筛选措施, 只对必要问题构建提示, 减少模型请求次数, 从而降低模型调用的消耗.

其次, LLM 在指令跟随方面存在不稳定性, 即不完全按照指令进行分析和输出, 我们认为这是导致分析结果不精确的重要原因. 对于配置间约束提取任务, 我们认为分析结果的正确性要求高于全面性要求, 因此为了降低 LLM 的不稳定性, 我们采取了数据预处理和配置标注、重点语句提示、添加少样本提示等多种措施, 来提高分析的精确率, 从而提高分析的正确性. 在这方面我们的方法依然有优化空间.

5 总 结

配置约束提取问题是软件配置研究领域的重点问题, 而配置间约束提取是其中相对复杂的问题. 本文提出了一种基于大语言模型提示工程的配置约束提取方法 LLM-Extractor, 通过自动化分析软件配置文档, 构建软件配置与软件功能关联图, 并使用该图谱中存在的多配置关联子图, 提取配置间约束关系. 为了评估 LLM-Extractor 的有效性, 本文基于 3 种流行的开源软件文档构建了评估数据集, 并基于该数据集验证了 LLM-Extractor 相对已有文本分析方法 PracExtractor 的有效性, 实验结果表明, LLM-Extractor 在配置间约束提取问题上相对 PracExtractor 具有显著优势, 在 $F1$ 分数上具有至少 43.4% 的提升, 这证明了 LLM-Extractor 的有效性. 进一步的消融实验表明, 相较于直接使用 LLM 分析整段配置文档, 使用多配置关联子图作为上下文能够显著提升 LLM 推断配置间约束的效果. 未来本文将进一步改进软件配置-功能关联图的构建方法, 同时将数据源扩展到软件日志、注释等多元自然语言信息, 抽取更多的关联关系, 提升配置间约束提取的效果.

References

[1] Wessels D, Nordström H, Jeffries A, *et al*. Squid: optimising Web delivery. 2024. https://www.squid-cache.org/

[2] Oracle. MySQL 5.7 reference manual. 2024. https://dev.mysql.com/doc/refman/5.7/en/server-configuration.html

[3] openGauss community. openGauss 5.0.0. 2024. https://docs-opengauss.osinfra.cn/en/docs/5.0.0/docs/DatabaseReference/guc-parameters.html

[4] Cloudflare. Understanding how Facebook disappeared from the Internet. 2019. https://blog.cloudflare.com/october-2021-facebook-outage/

[5] Tencent Cloud. Tencent Cloud April 8 incident postmortem and situation statement. 2024 (in Chinese). https://cloud.tencent.com/developer/article/2408984

[6] All GitHub services are experiencing significant disruptions—Incident report for GitHub. 2025. https://www.githubstatus.com/incidents/kz4khcgdsfdv

[7] Kiciman E, Wang YM. Discovering correctness constraints for self-management of system configuration. In: Proc. of the 2004 Int'l Conf. on Autonomic Computing. New York: IEEE, 2004. 28–35.

[8] Perrow C. Normal accidents: Living with high-risk technologies. Basic Books, 1984, 29(4): 630–632 [doi: 10.2307/2392945]

[9] Reason J. Human Error. Cambridge: Cambridge University Press, 1990. [doi: 10.1017/CBO9781139062367]

[10] Zhou SL, Li SS, Dong W, Wang J, Liao XK. Survey on software runtime configuration researches. Ruan Jian Xue Bao/Journal of Software, 2024, 35(1): 63–86 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6835.htm [doi: 10.13328/j.cnki.jos.006835]

[11] Xu TY, Zhang JQ, Huang P, Zheng J, Sheng TW, Yuan D, Zhou YY, Pasupathy S. Do not blame users for misconfigurations. In: Proc. of the 24th ACM Symp. on Operating Systems Principles. Farminton: ACM, 2013. 244–259. [doi: 10.1145/2517349.2522727]

[12] Liao XK, Zhou SL, Li SS, Jia ZY, Liu XD, He HC. Do you really know how to configure your software? Configuration constraints in source code may help. IEEE Trans. on Reliability, 2018, 67(3): 832–846. [doi: 10.1109/TR.2018.2834419]

[13] Chen QR, Wang T, Legunsen O, Li SS, Xu TY. Understanding and discovering software configuration dependencies in cloud and

datacenter systems. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 362–374. [doi: 10.1145/3368089.3409727]

[14] Zhang JL, Piskac R, Zhai EN, Xu TY. Static detection of silent misconfigurations with deep interaction analysis. Proc. of the ACM on Programming Languages, 2021, 5(OOPSLA): 140. [doi: 10.1145/3485517]

[15] Rabkin A, Katz R. Static extraction of program configuration options. In: Proc. of the 33rd Int'l Conf. on Software Engineering. Honolulu: IEEE, 2011. 131–140. [doi: 10.1145/1985793.1985812]

[16] Dong Z, Andrzejak A, Lo D, Costa D. ORPLocator: Identifying read points of configuration options via static analysis. In: Proc. of the 27th IEEE Int'l Symp. on Software Reliability Engineering. Ottawa: IEEE, 2016. 185–195. [doi: 10.1109/ISSRE.2016.37]

[17] Zhou SL, Liu XD, Li SS, Dong W, Liao XK, Xiong Y. ConfMapper: Automated variable finding for configuration items in source code. In: Proc. of the 2016 IEEE Int'l Conf. on Software Quality, Reliability and Security Companion. Vienna: IEEE, 2016. 228–235. [doi: 10.1109/QRS-C.2016.35]

[18] Zhang JQ, Renganarayana L, Zhang XL, Ge NY, Bala V, Xu TY, Zhou YY. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In: Proc. of the 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Salt Lake City: ACM, 2014. 687–700. [doi: 10.1145/2541940.2541983]

[19] Chen W, Qiao XQ, Wei J, Zhong H, Huang X. Detecting inter-component configuration errors in proactive: A relation-aware method. In: Proc. of the 14th Int'l Conf. on Quality Software. Allen: IEEE, 2014. 184–189. [doi: 10.1109/QSIC.2014.37]

[20] Santolucito M, Zhai EN, Piskac R. Probabilistic automated language learning for configuration files. In: Proc. of the 28th Int'l Conf. of Computer Aided Verification. Toronto: Springer, 2016. 80–87. [doi: 10.1007/978-3-319-41540-6_5]

[21] Santolucito M, Zhai EN, Dhodapkar R, Shim A, Piskac R. Synthesizing configuration file specifications with association rule learning. Proc. of the ACM on Programming Languages, 2017, 1(OOPSLA): 64. [doi: 10.1145/3133888]

[22] Bhagwan R, Mehta S, Radhakrishna A, Garg S. Learning patterns in configuration. In: Proc. of the 36th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: IEEE, 2021. 817–828. [doi: 10.1109/ASE51524.2021.9678525]

[23] Mehta S, Bhagwan R, Kumar R, Bansal C, Maddila C, Ashok B, Asthana S, Bird C, Kumar A. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In: Proc. of the 17th USENIX Symp. on Networked Systems Design and Implementation. Santa Clara: USENIX Association, 2020. 435–448.

[24] Tuncer O, Bila N, Duri S, Isci C, Coskun AK. ConfEx: Towards automating software configuration analytics in the cloud. In: Proc. of the 48th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks Workshops. Luxembourg: IEEE, 2018. 30–33. [doi: 10.1109/DSN-W.2018.00019]

[25] Xu XY, Li SS, Guo Y, Dong W, Li W, Liao XK. Automatic type inference for proactive misconfiguration prevention. In: Proc. of the 29th Int'l Conf. on Software Engineering and Knowledge Engineering. Pittsburgh: ksiresearch.org, 2017. 295–300. [doi: 10.18293/SEKE2017-072]

[26] Xiang CC, Huang HC, Yoo A, Zhou YY, Pasupathy S. PracExtractor: Extracting configuration good practices from manuals to detect server misconfigurations. In: Proc. of the 2020 USENIX Annual Technical Conf. USENIX Association, 2020. 265–280.

[27] Zhou SL, Liu XD, Li SS, Jia ZY, Zhang YL, Wang T, Li W, Liao XK. ConfInLog: Leveraging software logs to infer configuration constraints. In: Proc. of the 29th IEEE/ACM Int'l Conf. on Program Comprehension. Madrid: IEEE, 2021. 94–105. [doi: 10.1109/ICPC52881.2021.00018]

[28] Mandal S, Chethan A, Janfaza V, Mahmud SMF, Anderson TA, Turek J, Tithi JJ, Muzahid A. Large language models based automatic synthesis of software specifications. arXiv:2304.09181, 2023.

[29] Xiao CR, Xu SX, Zhang KP, Wang YF, Xia L. Evaluating reading comprehension exercises generated by LLMs: A showcase of ChatGPT in education applications. In: Proc. of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023). Toronto: ACL, 2023. 610–625. [doi: 10.18653/v1/2023.bea-1.52]

[30] Peng Y, Wang CZ, Wang WX, Gao CY, Lyu MR. Generative type inference for Python. arXiv:2307.09163, 2023.

[31] Zhao WX, Zhou K, Li JY, *et al*. A survey of large language models. arXiv:2303.18223, 2025.

[32] Brown TB, Mann B, Ryder N, *et al*. Language models are few-shot learners. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 1877–1901.

[33] Touvron H, Lavril T, Izacard G, *et al*. LLaMA: Open and efficient foundation language models. arXiv:2302.13971, 2023.

[34] Google. Gemini. 2024. https://gemini.google.com/

[35] Petroni F, Rocktäschel T, Riedel S, Lewis P, Bakhtin A, Wu YX, Miller A. Language models as knowledge bases? In: Proc. of the 2019 Conf. on Empirical Methods in Natural Language Processing and the 9th Int'l Joint Conf. on Natural Language Processing (EMNLP-IJCNLP). Hong Kong: ACL, 2019. 2463–2473. [doi: 10.18653/v1/D19-1250]

[36] Wei J, Wang XZ, Schuurmans D, Bosma M, Ichter B, Xia F, Chi EH, Le QV, Zhou D. Chain-of-thought prompting elicits reasoning in large language models. In: Proc. of the 36th Int'l Conf. on Neural Information Processing Systems. New Orleans: Curran Associates Inc., 2022. 24824–24837.

[37] Bian N, Han XP, Sun L, Lin HY, Lu YJ, He B, Jian SS, Dong B. ChatGPT is a knowledgeable but inexperienced solver: An investigation of commonsense problem in large language models. arXiv:2303.16421, 2024.

[38] Yang K, Tian YD, Peng NY, Klein D. Re3: Generating longer stories with recursive reprompting and revision. In: Proc. of the 2022 Conf. on Empirical Methods in Natural Language Processing. Abu Dhabi: ACL, 2022. 4393–4479. [doi: 10.18653/v1/2022.emnlp-main.296]

[39] Min BN, Ross H, Sulem E, Ben Veyseh AP, Nguyen TH, Sainz O, Agirre E, Heintz I, Roth D. Recent advances in natural language processing via large pre-trained language models: A survey. ACM Computing Surveys, 2023, 56(2): 30 [doi: 10.1145/3605943]

[40] Liu X, Ji KX, Fu YC, Tam W, Du ZX, Yang ZL, Tang J. P-tuning: Prompt tuning can be comparable to fine-tuning across scales and tasks. In: Proc. of the 60th Annual Meeting of the Association for Computational Linguistics (Vol. 2: Short Papers). Dublin: ACL, 2022. 61–68. [doi: 10.18653/v1/2022.acl-short.8]

[41] Liu PF, Yuan WZ, Fu JL, Jian ZB, Hayashi H, Neubig G. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Computing Surveys, 2023, 55(9): 195 [doi: 10.1145/3560815]

[42] OpenAI. Introducing ChatGPT. 2022. https://openai.com/blog/chatgpt

[43] Zhao ZH, Wallace E, Feng S, Klein D, Singh S. Calibrate before use: Improving few-shot performance of language models. In: Proc. of the 38th Int'l Conf. on Machine Learning. PMLR, 2021. 12697–12706.

[44] Su HJ, Kasai J, Wu CH, Shi WJ, Wang TL, Xin JY, Zhang R, Ostendorf M, Zettlemoyer L, Smith NA, Yu T. Selective annotation makes language models better few-shot learners. arXiv:2209.01975, 2022.

[45] Liu JC, Shen DH, Zhang YZ, Dolan B, Carin L, Chen WZ. What makes good in-context examples for GPT-3? In: Proc. of the 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures. Dublin: ACL, 2022. 100–114. [doi: 10.18653/v1/2022.deelio-1.10]

[46] Trajanoska M, Stojanov R, Trajanov D. Enhancing knowledge graph construction using large language models. arXiv:2305.04676, 2023.

[47] Kommineni VK, König-Ries B, Samuel S. From human experts to machines: An LLM supported approach to ontology and knowledge graph construction. arXiv:2403.08345, 2024.

[48] Carta S, Giuliani A, Piano L, Podda AS, Pompianu L, Tiddia SG. Iterative zero-shot LLM prompting for knowledge graph construction. arXiv:2307.01128, 2023.

[49] PostgreSQL global development group. PostgreSQL 12.22 documentation. 2024. https://www.postgresql.org/docs/12/

[50] Facebook. Faiss. 2024. https://github.com/facebookresearch/faiss

[51] OpenAI. GPT-4o-mini: Advancing cost-efficient intelligence. 2024. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[52] OpenAI. GPT-4o. 2024. https://openai.com/index/hello-gpt-4o/

[53] Soot. Soot—A framework for analyzing and transforming Java and Android applications. 2024. https://soot-oss.github.io/soot/

[54] Apache Hadoop. HDFS Architecture. 2018. https://hadoop.apache.org/docs/r2.9.2/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[55] Apache Hadoop. HDFS-default.xml. 2018. https://hadoop.apache.org/docs/r2.9.2/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml

附中文参考文献

[5] 腾讯云. 腾讯云 4 月 8 日故障复盘及情况说明. 2024. https://cloud.tencent.com/developer/article/2408984

[10] 周书林, 李姗姗, 董威, 王戟, 廖湘科. 软件运行时配置研究综述. 软件学报, 2024, 35(1): 63–86. http://www.jos.org.cn/1000-9825/6835.htm [doi: 10.13328/j.cnki.jos.006835]

作者简介

张添翼, 硕士生, 主要研究领域为云原生, 智能化运维.

周彤, 博士生, 主要研究领域为云原生, 智能化运维.

张晨曦, 博士, 副教授, CCF 专业会员, 主要研究领域为智能化运维, 云原生, 软件测试.

彭鑫, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为智能化软件开发, 云原生, 智能化运维, 泛在计算软件系统.