中国科学 : 信息科学 2026 年 第 56 卷 第 1 期: 59–75
SCIENTIA SINICA Informationis
论文

开发运行一体的复杂关键软件系统演化

丁博[1,2], 王怀民[1,2*], 米海波[1,3], 陈振邦[1,2], 付军[1,2], 张迅晖[1,2]

1. 复杂关键软件环境全国重点实验室, 长沙 410073
2. 国防科技大学计算机学院, 长沙 410073
3. 信息支援部队工程大学, 武汉 430000
* 通信作者. E-mail: hmwang@nudt.edu.cn

收稿日期: 2024–12–11; 修回日期: 2025–04–30; 接受日期: 2025–08–21; 网络出版日期: 2025–12–26

湖南省科技创新领军人才 (批准号: 2023RC1005) 资助项目

摘要 软件正在成为人类社会基础设施之一. 此类软件系统兼具 “复杂性” 和 “关键性” 的特点, 在其生命周期内, 需求和运行环境都在频繁变化, 传统软件工程方法很难为其提供有效支撑, 成为当前实践中的瓶颈问题. 要应对挑战, 就必须 “边建边用边升级”、让系统自身演化的速度跟上外部变化的速度. 然而, 复杂关键软件系统运行于高度管制环境, 具有服务质量可信确保、信息物理社会高度融合、利益相关方众多等特点, 其演化有着自身的独特规律, 很难简单地照搬互联网等领域的已有实践. 本文在分析复杂关键软件系统特点基础上, 提出开发与运行活动紧密结合、多利益相关方共同驱动的复杂关键软件系统演化方法 TrustieEvo, 阐述其需求持续管理、版本可信迭代、系统韧性增强 3 个方面核心使能机制, 给出相应软件平台设计. 方法支撑某真实复杂关键软件系统近 5 年的持续演化, 将 “用户新需求提出 – 软件新能力形成” 需求实现周期从 “年” 级别提升到 “周” 级别, 在实践中经受了高强度检验.

关键词 复杂关键软件系统, 高度管制环境, 软件演化, 软件代谢率, 软件韧性

1 引言

软件定义世界, 复杂关键软件系统正在成为社会生产生活的基础设施, 典型实例包括智慧城市系统、数字金融系统、指挥控制系统和电子政务系统等. 相较于传统分布式系统, 此类系统有两个突出特征: 一是关键性, 即其不再只是提高效率的辅助性 “工具”, 而是嵌入到人类社会生产生活之中, 成为具有基础能力、效用泛在、质量攸关特点的 “平台”; 二是复杂性, 即其兼具 “系统之系统”“信息 – 物理 – 社会” 系统和群体智能系统的特点 [1], 成为系统科学意义上的人造复杂系统.

软件领域对 “关键性” 已经有了许多实践. 在航空航天等任务关键系统中, 为了保证软件可信 (即软件行为符合预期 [2]), 基于 “还原论” 思想的软件开发方法 (如瀑布模型) 被广泛应用 [3], 并且特别强调过程严格可管控、阶段成果可验证、利益相关方责任划分清晰等经典软件工程原则. 然而, 此类方法仅适用于边界清晰、需求明确的系统. 当软件同时表现出 “关键性” 和 “复杂性” 特征之后, 新的挑战出现: 系统边界动态开放、由大量局部自治系统耦合关联而成, 软件系统直接承载着人类社会和物理世界的各种变化, 这两个因素都使得软件需求很

引用格式: 丁博, 王怀民, 米海波, 等. 开发运行一体的复杂关键软件系统演化. 中国科学: 信息科学, 2026, 56: 59–75, doi: 10.1360/SSI-2024-0371
Ding B, Wang H M, Mi H B, et al. Evolution of complex mission-critical software systems based on development and runtime cooperation. Sci Sin Inform, 2026, 56: 59–75, doi: 10.1360/SSI-2024-0371

难再事先完整描述、运行环境很难再事先精确预期 [1]. 这导致在今天许多复杂关键软件系统的实际建设过程中, 一方面, 人们对软件的能力预期很高; 另一方面, 越是核心、越是被广泛应用的软件组件, 部署后越容易出现用户满意度不高、适应性差的现象, 甚至出现 “软件发布之日就是淘汰之时” 的现象. 本文称此类现象为 “复杂关键软件系统悖论”. 例如, 英国国家医疗服务体系信息技术计划 (National Programme for IT, NPfIT) 总计支出超过 100 亿英镑, 但最终在开发 10 年后取消, 失败的一个重要原因是医疗系统需求、实践标准乃至外界政治因素等持续变化, 而项目本身难以适应这些变化 [4]; 美国防部科学委员会所发布报告 [5] 指出, 超过 60% 的国防部项目面临软件所带来的风险.

既然需求和环境的持续变化不可避免, 唯一的应对之道就是 “边建边用边升级”, 让软件系统自身变化的速度跟上外部变化的速度. 近年来, 在商业因素驱动下, 互联网软件的快速迭代已经取得一系列成功实践 [6, 7]. 但是, 对于在复杂关键软件系统中应用此类方法, 人们仍持有较大的疑虑 [8], “快速迭代” 和 “可信确保” 似乎存在天然隔阂, 一系列问题有待回答: (1) 相较于一般软件, 复杂关键软件系统到底有什么新特点, 会影响快速迭代演化的实施; (2) 已有的快速迭代演化方法需要进行什么样的改造和扩展, 才能适用于此类软件系统; (3) 快速迭代演化有可能会在可信确保方面引入不可控因素、造成灾难性后果, 如何增强用户、运维人员和其他利益相关方对新版本的信心.

本文在分析复杂关键软件系统特点基础上, 针对此类软件系统 “如何快速迭代演化” 和 “如何在快速迭代演化过程中保证可信” 两个核心挑战, 提出开发运行一体的复杂关键软件系统演化方法 TrustieEvo. 该方法的目标是同时提升复杂关键软件系统的寿命和代谢率: 通过在系统全生命周期内缩短 “用户新需求提出 – 软件新能力形成” 的需求实现周期 (即提升软件代谢率), 使软件具有生命力、实现长寿命高可信运行. 方法针对复杂关键软件系统服务质量可信确保、运行环境高度管制、信息物理社会紧密融合、利益相关方众多且权责分离等特点, 建立了由开发人员、运维人员、用户和权威管理方等利益相关者共同驱动的软件快速迭代演化技术框架[1)], 提出了由全生命周期需求持续管理、多方协同的版本快速可信迭代、开发与运行共促的系统韧性增强 3 部分组成的方法体系, 并给出了相应软件平台参考实现. 方法及平台应用于某真实复杂关键软件系统近 5 年的持续演化实践, 成功将软件需求实现周期从 “年” 级别提升到 “周” 级别, 显著提升了用户满意度.

本文后续内容安排如下: 第 2 节阐述复杂关键软件系统的概念和特性; 第 3 节分析现有软件技术面临的挑战; 第 4 节给出方法目标、总体框架及必要性测度指标; 第 5 节给出方法核心使能机制; 第 6 节给出支持方法落地实施的软件平台参考实现; 第 7 节结合实际案例, 分析 TrustieEvo 方法所获收益及经验教训; 第 8 节给出相关研究工作.

2　复杂关键软件系统

20 世纪 80 年代初, Lehman [9] 在对大型软件系统演化现象进行调研的基础上, 将软件分为 3 种基本类型: S 类型软件的功能由规格说明书 (specification) 形式化定义并严格推导, P 类型软件的需求来自具有一定不确定性的真实世界问题 (problem), E 类型软件则直接嵌入 (embedded) 到真实世界、驱动人类社会各类活动. 复杂关键软件系统可认为是 E 类型在今天的延续形态. 本文在文献 [1] 所提出的复杂软件系统概念基础上, 给出复杂关键软件系统的概念: 复杂关键软件系统是在社会生产生活中表现出能力基础、效用泛在、质量攸关特性的软件系统, 此类系统动态耦合大量异构且局部自治的子系统, 整个系统的行为难以通过各自治软件系统特征的简单叠加进行刻画.

上述概念强调了复杂关键软件系统的两个重要性质: 复杂性和关键性. 其中, 复杂性又可以展开为成员异质、行为涌现、边界开放、持续变化等子性质, 在文献 [1] 中已有详细论述; 关键性突出体现为: (1) 能力基础, 即它以常态化运行、联接多种要素及其外部环境的 “基础设施” 形态存在, 是社会生产生活得以正常运行的基石; (2) 效用泛在, 即软件功能受众并不局限于目标系统中特定要素, 也不仅仅是提升特定维度的运行效率, 而是在

1) TrustieEvo 强调软件开发及运行过程中多个利益相关方 (而不仅仅是开发和运维部门) 的高效协同, 这是方法前缀为 “开发运行一体” 而非 “开发运维一体” 的原因.

整个系统层面上有着普适的赋能作用; (3) 质量攸关, 即软件的作用域不仅包括信息系统自身, 而且还涵盖了与软件系统相连接的物理系统和社会系统, 能否提供可信服务, 将直接影响真实世界系统可靠运行, 乃至于人类生命财产安全.

需要特别指出的是, 复杂关键软件系统是 “定义” 社会生产生活各类复杂关键系统 (如社会经济系统、大型复杂装备等) 的软件系统, 上述复杂性和关键性是对社会生产生活进行 “软件定义” 的必然结果. 传统人工制品的功能固化在其硬件设计中, 一旦定型后就很难再调整. 与此不同, 软件作为一种特殊的人工制品, 通常表现为程序和数据, 理论上可以随时随地修改、裁剪和定制, 甚至在运行时进行升级替换. 这种灵活性与软件天然的丰富表达能力相结合, 使得软件有可能为社会生产生活中各类装置/系统赋予更灵活的使用方式和新价值, 以及能力在未来持续扩展的可能性.

以下为两个复杂关键软件系统案例.

案例1 (民用航空软件系统)　航空器的正常飞行依赖于机载综合电子软件、空地通信软件、地面空管软件等一系列子系统动态耦合关联而成的软件系统, 这些子系统隶属于不同的管理域、部署在不同计算设备、由不同人员开发和维护, 系统正确性直接关系到人的生命安全和经济的正常运转.

案例2 (有人/无人协同软件系统)　在软件的支持下, 有人系统和无人系统之间、各个无人系统之间既各自进行相对独立的组织、决策、规划、控制和感知等, 又通过自发且平等的交互共融, 实现共同目标的群体行为 [10], 如灾难救援、太空探索和反恐安全等.

3　现有软件技术面临的挑战

为了走出 20 世纪 60 年代的 “软件危机”, 人们从现代大规模工业生产中借鉴成功经验, 指出 “软件开发应是类似工程的活动” [11], 提出了以瀑布模型为代表的经典软件工程方法. 近年来, 互联网等领域对软件特有的规律有了进一步的认识, 在快速迭代演化方面开展了一系列成功实践 [6,7]. 上述方法推动了过去 50 余年中软件产业乃至信息产业的繁荣. 然而, 它们均有其各自适用场景和固有的局限性.

3.1　经典软件工程方法的局限性

如前节所述, 复杂关键软件系统是 “定义” 社会生产生活各类系统的软件. 很多时候, 如何进行软件 “定义”、如何利用信息化手段提高效率, 这些问题本身就在应用域中没有明确答案, 处于 Cynefin 决策框架中的难解 (complicated) 域 [12]. 同时, “定义” 也意味着人类社会和物理世界的变化将由软件直接承载, 在软件整个生命周期内, 需求和环境的频繁变化不可避免. 这打破了经典按部就班、计划驱动的软件工程方法的一系列假设 [1], 导致项目很容易陷入前文所提的 “复杂关键软件系统悖论”, 突出表现在如下两个方面.

(1) 用户需求实现的严重滞后. 传统 “需求分析 – 设计实现 – 定型发布” 循序渐进的大型软件开发周期长达数年, 软件系统最终是否可用好用, 在很大程度上依赖于初始阶段的需求分析结果, 在该阶段之后变更需求会带来巨大的成本开销和进度延迟, 这从根本上决定了经典软件工程方法并不适用于复杂关键软件系统. 以国防领域为例, 20 世纪 90 年代中期, 美国防部即确立了以瀑布模型为基础的软件开发与文档标准 MIL-STD-498 (以及后续民用领域的 EIA/IEEE J-STD-016 标准等) [13], 并一直沿用 20 余年. 发表于 2019 年的文献 [14] 指出, 遵循相应软件采办方法, 美国防部大型软件通常需要 3∼10 年才能交付, 其中从合同授予到软件测试两个里程碑之间通常需要 5∼7 年, 导致软件实现的是数年前的需求, 软件能力很难跟上作战样式和场景的变化.

(2) 环境相容的高度不确定. 经典软件工程方法要求对软件所依赖的运行环境给出明确假设, 然后在开发阶段即构建相应环境, 通过测试、形式化验证等手段来保证软件可信. 然而, 复杂关键软件系统通常是由大量系统/子系统所组成的复杂体系, 各个软件系统/子系统不是独立的 “烟囱”, 它需要与其他软件、外部服务、硬件平台、网络设备和物理传感器/效应器等交互, 才有可能正常运行, 进而通过 “联接” 形成体系支撑、发挥效用. 由于规模庞大、建设周期长等原因, 这些环境因素很可能由其他利益相关方提供, 同样在 “边建边用边升级”, 存在大量内部和外部不确定性 [15]. 软件很难在开发阶段精确预期其环境相容能力, 导致部署后问题频出、无法达到

设计目标.

3.2　互联网新型软件开发方法的局限性

近年来, 在商业因素的驱动下, 互联网等领域已经充分认识到快速满足用户需求的重要性, 涌现了 Scrum、看板 (kanban) 等敏捷开发方法 [6], 以及支撑软件版本快速形成能力的开发运维一体化 (DevOps) [7] 等方法. 但是, 相较于社交群组、视频点播和新闻资讯等互联网一般商业软件, 复杂关键软件系统虽然同是网络化的大规模软件系统, 但具有如下 5 个方面明显的区别.

(1) 服务质量从尽力而为到可信确保. 互联网发展之初是 "信息互联网", 其底层设计采用了尽力而为思想 [16], 当前大多数互联网商业软件也继承了这一思路, 即使要支撑关键应用场景, 也通常会以最终用户许可协议等形式进行免责. 然而, 由于复杂关键软件系统的 "关键性" 特征, 在很多场景下很难通过免责协议等方式绕过监管, 能否提供 "可信确保" 的服务, 将直接影响真实世界系统可靠运行, 乃至于人类生命财产安全.

(2) 软件开发及运行活动受高度管制. (i) 与商业化软件系统的实际收益可通过市场来衡量不同, 复杂关键软件系统的开发和运行活动都在高度管制环境 (highly regulated environment, HRE) [17] 中, 必须满足特定权威管理方所提出的需求、标准规范和技术体制, 达到权威方所规定的质量要求; (ii) 软件构建过程中所依赖的生态也受到权威管理方约束, 开源代码等的复用受到一定限制; (iii) 软件采办活动中市场机制所能起到的优胜劣汰作用相对滞后.

(3) 开发、运维和使用 3 个部门客观分离. (i) 由于管理体制、数据隐私、安全保密等客观原因, 此类软件系统 (或其子系统) 多数仍以 "交钥匙" 形式交付给专职运营单位, 很难直接套用互联网公司内部开发运维一体化的思想; (ii) 在规模较大、保密要求更高的系统中 (如国防软件系统), 甚至其多个开发商的开发环境是物理隔离的, 开发环境和真实运行环境是物理隔离的 [17]; (iii) 此类系统中使用部门承载关键业务, 往往相对具有话语权, 很难培养或引导其使用习惯.

(4) 信息 – 物理 – 社会 3 个维度紧密融合. (i) 复杂关键软件系统通常与物理世界和人类社会紧密耦合, 软件系统各个利益相关方同时是渐进式信息化进程的参与者, 导致大量需求延迟到软件发布甚至运行阶段才提出, 并且可能不断变化; (ii) 互联网软件服务通常可以集约化部署在一个或几个数据中心, 通过网页或 APP 提供服务, 但复杂关键软件系统需要 "嵌入" 到人类社会和物理世界、驱动人类社会运转, 通常采用云际 [18] 和 "云 – 边 – 端" 等灵活部署形态.

(5) 系统规模带来大量参与者及内在冲突. 复杂关键软件系统通常是大规模分布式系统, 其建设通常经历数年甚至数十年的 "边建边用边升级" 过程, 因此会存在大量利益相关方, 且这些利益相关方之间技术体制、管理策略等的冲突十分普遍, 涉及广地域、多形态的节点. 例如, 美国空军先进作战管理系统 (Advanced Battle management System, ABMS) 仅 2020 年的软件开发活动就涉及 70 个工业开发团队、65 个政府开发团队、28 条产品线中的 33 个平台, 以及数以百计的新研/遗留系统 [19].

上述区别决定了我们可以借鉴互联网软件开发运维一体化 (DevOps) 方法 [7] 等的思路, 但不能简单照搬现有实践, 主要挑战归结起来有两点: (1) 如何快速迭代演化, 即在软件开发和运行活动受到高度管制、开发/运维/使用 3 个部门客观分享、信息 – 物理 – 社会 3 个维度紧密融合、存在大量参与者和内在冲突的情况下, 软件系统的快速迭代演化需要什么样的方法和基础设施, 以解决 "高度管制" 与 "快速迭代" 如何不产生矛盾、大量利益相关方如何在迭代活动中无缝协同、如何有效应对 "边建边用边升级" 过程中暴露出的大量需求、迭代活动如何适应 "云 – 边 – 端" 部署架构、如何跨多个管理域推动快速迭代等一系列问题; (2) 如何在快速迭代演化过程中保证可信, 即如何在此类软件系统快速迭代演化的同时, 仍能够提供严苛的软件可靠性、可用性等保证, 并解决用户、运维人员和权威管理方对快速迭代过程中软件质量可能退化的疑虑.

4　TrustieEvo 方法概述

软件是由人开发的. 复杂关键软件系统要满足各种运行时变化的需求、适应设计时未预期的环境, 可以在

一定程度上借助机器智能的力量, 但更重要的是, 要将各个利益相关方 (开发人员、运维人员、用户和权威管理方等) 系统化地编织起来, 共同驱动软件系统的快速迭代演化, 这是 TrustieEvo 方法在架构设计层面上的基本出发点. TrustieEvo 方法并不追求在开发阶段 “毕其功于一役”, 而是强调开发活动不是软件发布后就停止 (或者只是进入维护阶段), 在宏观上将继续与软件运行活动伴随式展开、由多个利益相关方共同推进, 在微观上形成多个并发推进、快速迭代的 “用户新需求提出 – 软件新能力形成” 周期, 并且持续地映射到已部署甚至已在运行的软件系统之上, 驱动软件的持续演化.

4.1　方法目标

在当前建设实践中, “软件系统上线运行 – 用户不满意 – 重新立项从头再来” 的反复循环并不鲜见, 它是 “复杂关键软件系统悖论” 的必然结果, 在造成巨大人力物力浪费的同时, 系统能力也长时间在原地踏步. 为打破这种循环, TrustieEvo 方法强调复杂关键软件系统应当是长寿命并持续成长的, 并且这种长寿命不是靠行政命令、不是靠牺牲用户利益来维持的. 此类软件系统即使在部署后, 也应当持续接收用户新需求, 并通过缩短 “用户新需求提出 – 软件新能力形成” 的时间周期 (即提升软件代谢率) 来跟上需求和环境的变化, 使之能够具有长久的生命力.

换言之, TrustieEvo 方法的目标是同时提升软件寿命和软件代谢率. 其中, 软件寿命可定义为该软件首个实例交付使用时间与该软件所有实例退出使用的时间之差. 对于软件代谢率, 与生物体通过新陈代谢实现能量获取、维持活力和排出废物类似, 软件的 “新陈代谢” 是指其软件交付使用之后, 从外界 (如开发人员) 汲取力量、适应用户需求 (包括环境) 变化的活动, 因此代谢率可使用 “用户新需求提出 – 软件新能力形成” 的需求实现周期来衡量. 二者呈反比关系, 即代谢率越高, 需求实现周期越短.

定义1 (需求实现周期)　给定软件 s, 在交付使用后某个时间周期 T 内共新增并实现 n 条用户需求, 记第 i 条用户需求提出时间为 $t_{\text{requirement}[i]}$, 包括该功能实现的软件新版本上线运行的时间为 $t_{\text{running}[i]}$, 则该时间周期 T 内的需求实现周期 $M_s(T)$ 可记为

$$M_s(T)=\frac{1}{n}\sum_{i=1}^{n}(t_{\text{running}[i]}-t_{\text{requirement}[i]}). \tag{1}$$

4.2　方法总体框架

TrustieEvo 方法的总体框架如图 1 所示. 方法所作用对象是在广地域范围内、以云际或 “云 – 边 – 端” 等形态灵活部署的复杂关键软件系统. 方法通过引入专门技术框架和被称为 “协调者” 的专门技术力量, 将高度管制环境下开发、运维、用户等利益相关方系统化地编织到一起, 驱动复杂关键软件系统的快速迭代演化活动; 通过开发阶段多种可信保证措施和系统层面上、开发运行共促的韧性增强方法, 来兼顾 “快速迭代” 与 “可信保证”. TrustieEvo 方法框架具体可以分成如下 3 个层次.

(1) 群智驱动的软件版本快速发布. 软件版本的快速发布主要由软件开发部门和协调者负责, 由需求管理、开发、持续集成、自动测试、快速部署等环节组成 (图 1 左侧), 其目标是形成软件 “需求持续到达 – 群智协同开发 – 版本持续发布” 的高效通路, 从而在初始阶段形成软件最小可用版本, 以及在持续演化过程中支持软件版本的可信快速迭代. 新版本的发布将伴随相应的可信证据、运维知识和应急预案, 协调者可以基于这些知识为运维部门和用户部门提供常态化支撑.

(2) 大数据和运维人员驱动的现场适应调整. 现场适应调整由环境和状态监控、运行数据汇聚分析、运行调整决策、软件系统在线调整等环节组成 (图 1 中右下角框), 构成与软件服务伴随运行的 “免疫” 系统, 实现运行现场的可感知、可修复、可预警、可演化, 显著提升软件服务应对预期或非预期场景变化的 “韧性”. 其中, 运行调整决策可以在监控大数据的驱动下由软件自身根据预设规则或内置算法进行自主决策 [20], 也可以由运维人员综合软件运行状态做出决策. 这一过程中, 感知、修复和适应调整所能达到的能力上限受软件已部署版本能力的限制.

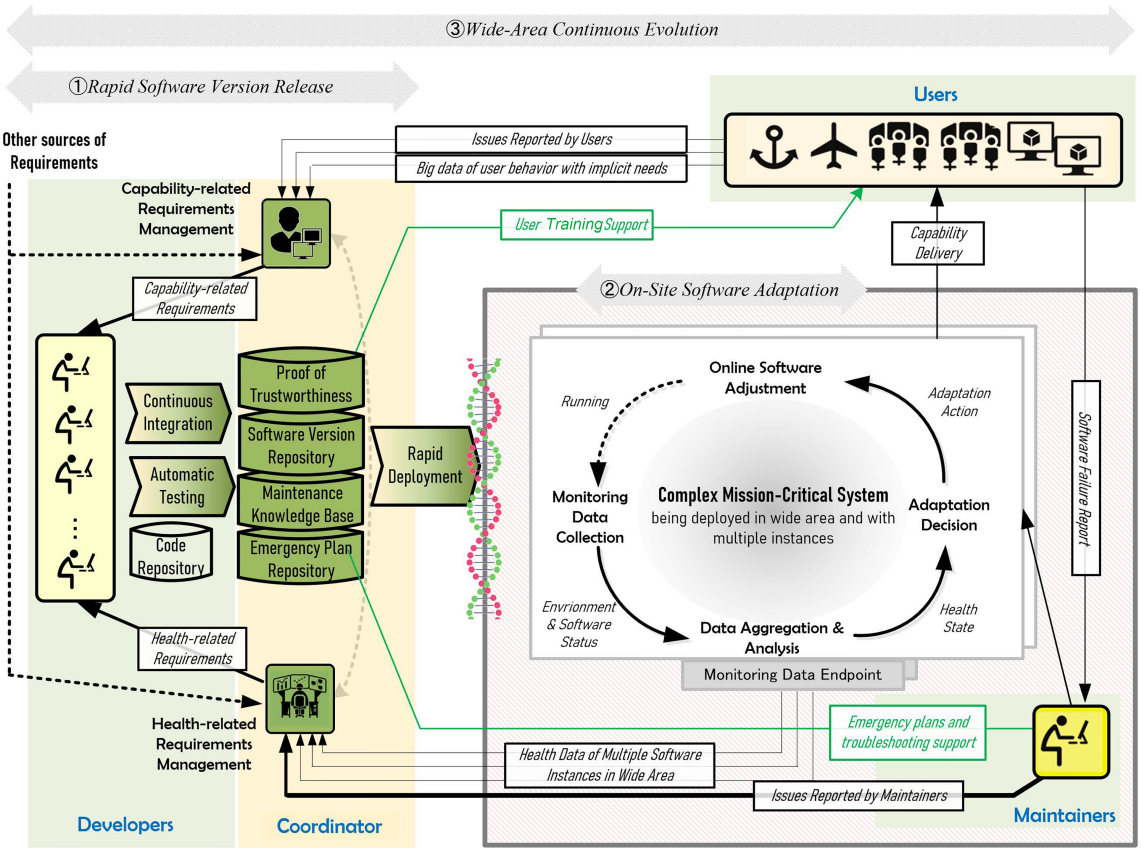

图 1　(网络版彩图) **TrustieEvo 方法总体框架.**

Figure 1　(Color online) Overall framework of the TrustieEvo approach.

(3) 多利益相关方共同驱动的广域持续演化. 这一过程由软件开发部门、运维部门、用户部门和专职的协调者共同驱动 (即图 1 中所有实体), 在广地域范围、较大时间尺度上实施, 是 TrustieEvo 方法的核心, 其目标是满足用户在使用过程中涌现的新需求、增强系统的韧性等. 在相应软件平台 (参见第 6 节) 的支持下, 广地域范围内多个复杂关键软件系统节点的运行状态数据将被汇总, 这些数据与软件运维部门反馈的软件问题一起提交给协调者, 形成代表系统健康运行和适应环境能力的健康性需求, 再由协调者反馈给开发部门; 同时, 匿名化的用户使用数据 (如使用习惯、偏好等) 将被汇总, 这些数据与用户在使用过程中主动提交的需求建议一起, 在软件平台支持下反馈给协调者, 形成能力性需求列表; 软件开发部门在获得能力性需求和健康性需求后, 开启一轮新的迭代, 通过群智驱动的软件版本快速发布, 形成满足用户需求的新版本并部署.

在上述过程, 特别是版本持续演化过程中, 协调者扮演了十分重要的角色, 除了联接和协调开发、运维、用户三者外, 还充当如下一些角色: 软件部署及运行态势的全局掌控者, 需求的汇聚、发现和管理者, 软件版本和知识的管理者, 软件可信证据的累积方, 软件部署前的可信性评估者, 远程推送部署的具体实施方. 事实上, 它在一定程度上发挥了软件开发活动相关的权威事实来源 (authoritative source of truth, ASOT) [21] 作用. 在某些场合下, 协调者可以由软件开发或运维部门来兼任. 但我们前期实践表明, 相对独立、职责明确、得到权威管理方授权的协调者对于方法效用发挥十分关键.

4.3　必要性测度指标

并非所有软件都需要快速迭代演化. 例如, 对于内嵌到航天器中的飞控软件, 由于其需求十分明确, 且运行于相对封闭环境、作用于行为可精确建模的物理装置, 其快速迭代的必要性并不明显. 而对于广域部署的智慧城市、数字医疗、指挥控制等大型分布式系统, 则有着明显的快速迭代必要性. 表 1 给出应用 TrustieEvo 方法

表 1　TrustieEvo 方法应用的必要性测度指标.

Table 1　Metrics of evaluating the necessity of application of the TrustieEvo.

	Metrics	Necessity: high	Necessity: medium	Necessity: low
1	Human in business path	At the core step of the critical path	On the critical path, but not in the core step	Not on the critical path
2	Physical devices in path	Physical device behavior is unpredictable and in the core step of the critical path	Physical device behavior is unpredictable, on the critical path but not in the core step	Not involving physical devices or not on the critical path, or its behavior is strictly predictable
3	Requirements change frequency	Constantly changing and difficult to predict	Constantly changing, but more predictable	Unchanged, or only minor expected changes
4	Software upgrade frequency	Days or weeks	Months	Years
5	Time to fix bugs/vulnerabilities	Hours	Days	Months
6	Deployment environment	Diverse and difficult to predict	Diverse and easy to predict	Homogeneous
7	Deployment scale	Large-scale in a wide area	Large-scale in a few sites	Small-scale
8	Deployment time	Hours	Days	Weeks
9	Expected failure recovery time	Minutes	Hours	Weeks
10	Tolerable upgrade rollback rate	$<15\%$	$15\%\sim30\%$	$>30\%$

的必要性测度指标. 这些指标多数是定性的, 仅原则性给出衡量一个系统是否有必要快速迭代的基本维度及指导性测量方法. 我们前期实践中, 通常认为 2 项以上指标为 “高” 或 5 项以上指标为 “中”, 即有应用 TrustieEvo 方法的迫切需求.

5　TrustieEvo 方法核心机制

第 4.2 小节给出了 TrustieEvo 方法总体框架, 这一框架要落地实施, 涉及全生命周期需求持续管理、多方协同的版本快速可信迭代、开发与运行共促的系统韧性增强 3 个方面核心使能机制. 本节简述我们前期在方法框架落地方面的初步研究成果, 以及对相关挑战的进一步认识.

5.1　全生命周期需求持续管理

复杂关键软件系统采用 “边建边用边升级” 的模式, 本质上是在尽早满足用户急迫需求的同时, 汇聚和激发系统研发、建设、运用、管理和维护各利益相关方的群体智慧, 深化对信息化条件下如何开展特定领域业务活动的认知, 凝练合理需求并不断快速验证. 要实现这一目标, 前提是需求能够从各利益相关方持续、快速、尽可能不失真的反馈给开发团队. 然而, 在已有实践中, 对于广地域范围、以云际或 “云 – 边 – 端” 等形态部署的大型系统, 其需求采集通常需要通过行政管理手段, 驱动众多利益相关方之间的协同; 需求汇聚则更是一个繁杂的手工过程, 时间可能长达数年 [1]. TrustieEvo 方法的全生命周期需求持续管理机制 (图 2) 通过在软件基础设施层面引入多条显式/隐式需求采集回路, 将原来需要采集的手工过程变成与软件系统伴随运行的自动/半自动化过程, 并在这一过程中强调 “数据即需求”, 发掘各类运行监控数据、用户行为数据等大数据中隐含的需求, 进而由需求管理人员对所采集需求 “去粗取精、去伪存真”、分解为具体开发任务. 这一机制是图 1 中能力性需求和健康性需求管理过程的细化. 与敏捷需求管理 [22] 等工作仅关注开发阶段需求如何逐步求精不同, 它是各利益相关方高度参与, 并且贯穿至软件整个生命周期的. 具体而言, 该机制包括如下内容.

(1) 原始需求数据的广地域采集. TrustieEvo 方法强调 “数据即需求”, 在广地域范围内建立软件 “边建边用边升级” 过程中的需求反馈回路, 这些需求原始数据一部分以用户或运维人员通过在线系统、以自然语言形式提报, 一部分则来自自动采集、隐含软件需求的用户行为数据和软件运行监控数据. TrustieEvo 方法将原始需求

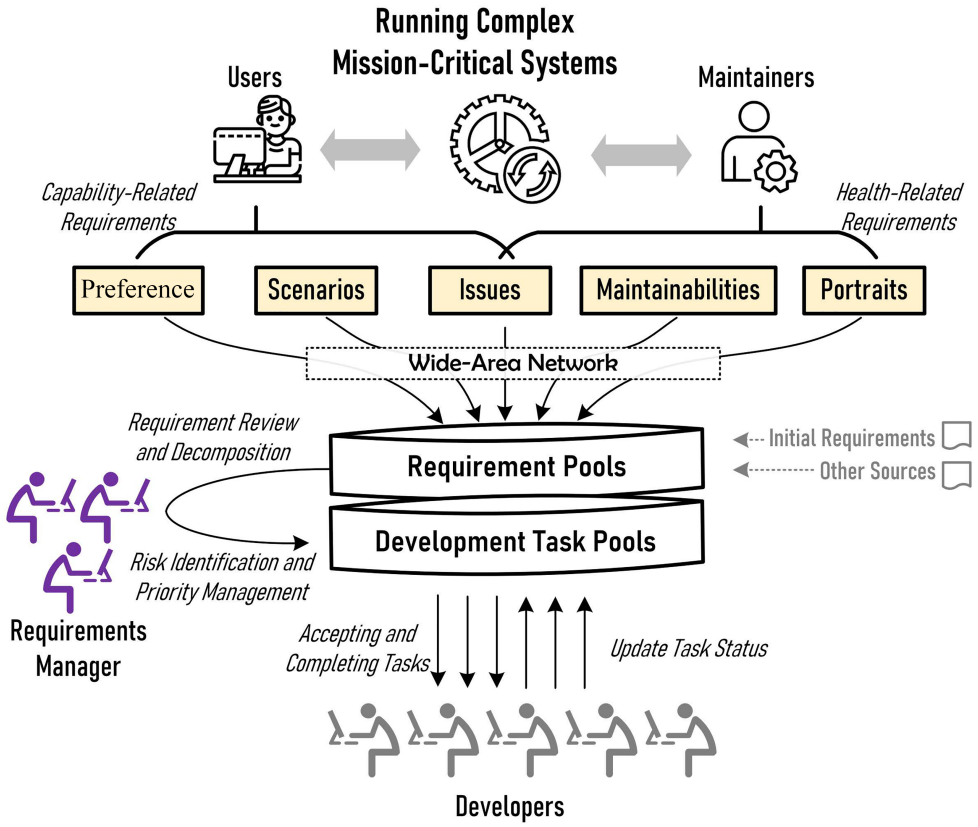

图 2　(网络版彩图) 全生命周期需求持续管理.

Figure 2　(Color online) Continuous requirements management throughout lifecycle.

数据分为能力性和健康性两类. 其中, 能力性需求是用户所直接感知的需求 (如等 “应当增加某功能, 其使用场景如下 . . . ”“ 某菜单经常卡顿, 应当优化”), 具体包括: (i) 场景 (scenarios), 即用户以自然语言给出的基于场景、较为完整的需求描述, 通过社区采集、手工提报等方式获取; (ii) 疑修 (issues), 即用户使用过程中以自然语言给出的相对零散需求表达, 通过用户界面直接反馈、社区采集、手工提报等方式获取; (iii) 偏好 (preferences), 即蕴含使用偏好、常见使用流程等的用户行为大数据, 通过后台行为监测系统自动采集. 健康性需求是运维活动暴露出来的需求 (如 “在自主平台上某功能失效”“应当具备统计当日某业务数据能力”“集群应具备弹性扩展能力”等), 具体包括: (i) 疑修, 与健康性需求中疑修不同的是, 它是由运维人员提出的, 是以自然语言给出的对软件系统缺陷修正、能力/性能优化、环境适应等的需求表达; (ii) 可维 (maintainabilities), 是以自然语言给出的软件维护能力 (如对运维监控设施) 需求表达, 可以通过运维界面直接反馈、社区采集、手工提报等方式获取; (iii) 画像 (portraits), 即表征健康性的软件实时运行监控大数据, 由伴随运行的监控系统自动采集.

(2) 基于大数据的需求汇聚与分析. 各类需求原始数据被汇聚到图 1 中的协调者, 针对不同类型数据, 需求汇聚的方式方法是不同的. 针对自然语言表达的需求, 重点是对原始数据进行合并去冗、“去粗取精、去伪存真”, 然后进行必要分解, 产生非耦合、非歧义、开发人员可理解的任务集; 针对行为数据和运行监控数据, 重点是从大数据中挖掘隐藏的功能、性能、可靠性、可用性、可维护性等需求. 由于广域采集数据的规模庞大, 上述工作可以在机器智能手段辅助下进行. 实践表明, 在需求持续到达、原始需求表述不精准的情况下, 类似于文献 [23] 的方法是有效的, 但需要引入私有知识库、知识图谱等形式, 为大模型补充复杂关键软件系统所在的领域知识.

(3) 需求实现的细粒度多轨并发跟踪. “开发与运行并行” 最终在微观层面上表现为由不同需求驱动、在不同时刻启动的多个 “开发 – 测试 – 集成 – 部署” 活动, 这些活动可能多线并行推进, 这与传统软件只有一条 “开发 – 测试 – 集成 – 部署” 时间线不同. 因此, 需要在开发环境层面建立对多个需求实现状态进行跟踪的手段, 并尽可能将需求对应的开发任务解耦, 以使得有可能生成软件阶段性可部署版本. TrustieEvo 方法在软件平台的

群智开发部分中引入了专门的需求持续跟踪管理部件 (参见第 6 节), 通过将需求动态区分为基础性需求 (不可解耦的基础能力) 和增量性需求 (可解耦、可有选择组合的能力)、基于需求 Id 的全流程跟踪及查询预警、需求优先级自动/手工调整等机制, 来支撑需求实现的细粒度多轨并发跟踪.

5.2　多方协同的版本快速可信迭代

在需求纳入开发团队的管理之后, 需要尽快在软件新版本中实现相应能力, 以保证 “用户新需求提出 – 软件新能力形成” 的需求实现周期尽可能缩短. 这一过程中的核心挑战在于两个方面: 如何快速开发出满足用户新需求的版本; 如何在版本迭代过程中保证软件可信, 并给权威管理方和用户对软件新版本以信心.

(1) 基于群智范式的软件版本迭代. 软件开发本质是对 “群体智能” 的关注 [24], 如何在大型软件系统的开发活动中充分利用群体的力量, 这一问题长期以来备受关注. 典型的反例是软件工程领域早期提出的 Brooks 法则 [25]: 增加人员到一个已经延误的项目, 只会使进度更加落后. 针对如何在群智激发 (扩大群体规模) 和汇聚 (产出可信产品)、在需求的确定性和不确定性之间找到平衡点这一问题, 我们在基于群体智能的可信软件大规模协同开发方法方面展开探索, 在群体大规模协作开发机制、软件资源开放共享和推荐复用、持续性评估和软件可信保证、协同开发基础环境构建等方面取得突破 [2, 26, 27]. 在此基础上, 吸收开源范式和工程范式优点, 提出软件开发群智范式 [24], 围绕该范式下的新型软件开发机理、方法技术和运行机制开展了研究, 并针对复杂关键软件系统在高度管制环境运行、利益相关方众多等特点, 构建了基于可控开源平台、支撑版本快速迭代的群智开发环境, 以及基于容器的快速构建和远程部署机制. 由于容器技术的隔离性、标准化、可分层等特点, TrustieEvo 方法鼓励新版本以容器镜像 (及编排后的镜像组) 形式发布, 我们对基于机器学习方法的容器构建时间预测、容器构建过程中故障根因分析等展开了深入探索, 对提升软件版本迭代速度起到了重要作用 [28~30].

(2) 版本迭代过程中的软件可信保证. 由于引入新代码或修改已有代码都有可能导致软件系统不可信, 开发人员求 “新”、运维团队求 “稳” 在实践中成为阻碍软件新版本上线的主要矛盾之一 [7]. 这一矛盾在复杂关键软件系统中变得更为激烈, 一方面, 此类软件系统利益相关方众多, 用户、运维人员、权威管理方等任意一方对新版本没有信心都可能成为新版本上线的阻力; 另一方面, 在经典的工程化方法中, 每个版本出厂前的配置项和系统测试有可能长达数周甚至数月, 其速度很难支撑软件快速迭代演化的目标. TrustieEvo 方法通过如下两个方面来应对挑战: (i) 测试过程的全流程自动化, 在借鉴民用领域持续集成/自动测试/快速部署成功实践基础上, 实现对复杂关键软件系统功能、性能、压力、界面、接口、安全性 (抗攻击)、环境适应性 (抗恶劣环境) 的全面测试流水线; (ii) 可信保证知识的复用, 即每一次迭代尽量复用上一个版本的可信保证知识及证据, 尽可能只关注增量部分, 例如, 我们提出了混合回归测试用例选择技术 [31], 通过增量式构建、有效融合多粒度的测试用例影响域信息, 在测试时仅执行受版本迭代影响的测试用例, 从而避免每次版本迭代都重复执行所有测试, 可有效提升测试流水线的执行效率.

软件可信的另一个重要方面是给权威管理方和用户对新版本以信心. 在当前任务关键软件系统的实践中, 这主要通过第三方测试和专家评审 (如文档评审) 等方式实现. 但是, 当环境动态变化、版本迭代速度达到 “周” 或者 “天” 级别时, 上述方式已无法有效实施. TrustieEvo 方法强调权威管理方的角色定位应从每个版本的 “能力评审” 向 “团队准入” 转变, 从 “产品把关” 向 “过程监管” 转变, 并且通过从传统计价 (如功能点估算) [32] 向使用价值计价的计价方式变革, 激励开发团队开发高质量产品; 同时强调规范化复杂关键软件系统各参研团队的开发环境、测试环境和相关工具链, 并通过持续文档 [33] 等方式实现过程存证留痕, 从而有利于过程的监管.

5.3　开发运行共促的系统韧性增强

第 5.2 小节中的可信保证机制仍是以孤立的眼光去看待复杂关键软件系统中的各个软件实体. 然而, 复杂关键软件系统往往是 “系统之系统”, 单个组分的可信远远不够. 例如, 一个广域数据传输系统的功能和性能是否正常, 不仅依赖于其本身业务逻辑是否正确, 往往还依赖网络环境、安防设备和数据公共存储服务等是否正常运行. 体系层面上的问题需要寻求体系层面上的解决方案, 开发运行共促的系统韧性增强机制即聚焦于系统/子系统层面上的可信问题. 韧性 [34] 是指软件系统在自身或外部环境出现预期或非预期危机时, 在整体上仍能持续提

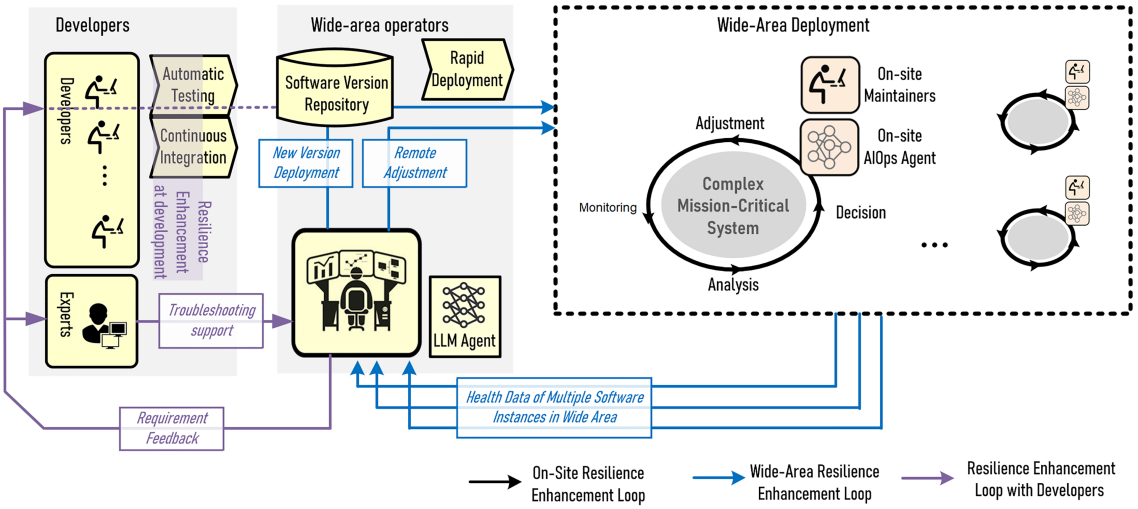

图 3　(网络版彩图) 开发运行共促的系统韧性增强.

Figure 3　(Color online) System resilience enhancement based on development and runtime cooperation.

供服务, 并且能够从危机中快速恢复的能力. 韧性不以追求软件系统消灭所有问题隐患为前提, 而是在接受危机必然存在的背景下, 通过健壮的系统架构设计、必要容错冗余、持续成长与演化等机制, 实现可感知、可修复、可预警和可演化, 进而实现恶劣条件下的 “顽存”. 这一思路突破了经典软件工程方法中单纯追求逻辑正确性的思路.

图 3 给出了 TrustieEvo 方法中的韧性增强回路, 包括现场回路、广域回路和厂商回路, 这些回路是 TrustieEvo 方法框架 3 个层次在韧性增强活动中的具体映射. 根据发生地点不同, 具体的韧性增强活动包括如下内容.

(1) 开发阶段韧性增强. 开发阶段韧性增强包括韧性体系结构设计和韧性试验. 韧性体系结构设计方面, TrustieEvo 方法提出基础设施不变、业务逻辑可变、运行形态可编排、内部状态可观测、应用需求可反馈、不同尺度可运控的体系结构风格约束, 给出单个结点容器加固、广域范围云边协同的体系结构参考实现; 韧性试验方面, TrustieEvo 方法将混沌工程 [35] 的思想引入复杂关键软件系统的模拟部署环境, 通过模拟内部软件组件、外部软件服务、网络环境等的随机故障和有意攻击, 来发现韧性问题.

(2) 运行现场韧性增强. 运行现场韧性增强通过构建与软件服务伴随运行的 “监控 – 分析 – 决策 – 调整” 现场回路实现, 该回路通过探针、日志、请求调用链等手段采集环境和软件运行状态, 及时发现故障和隐患, 并在发生问题时由现场运维人员或机器智能 (如预定义规则或机器学习模型) 实时介入和调整, 这一回路相当于软件运行时的 “免疫系统” 或 “疾控系统”.

(3) 广域开发运行一体韧性增强. 复杂关键软件系统往往采用云际或 “云 – 边 – 端” 形式灵活部署, 同一软件服务在不同环境、不同地点部署有多个副本, 不同的软件服务通过网络互联、共同支撑特定业务, 其运行状态数据存在内在联系. 广域开发运行一体韧性增强首先构建广域范围内运行数据的系统化采集和汇聚手段, 进而利用运行数据多来源、多副本、相互关联等特性, 深度发现故障和隐患, 及时对运行现场如何处置问题给出指导; 更进一步地, 软件厂商被体系性地纳入韧性增强回路中, 一方面以开发团队专家形式介入问题处置流程, 另一方面也得以在必要时通过发布新版本并推送部署的形式增强软件韧性.

6　TrustieEvo 软件平台

软件平台 [36] 针对一类软件的开发与运行活动, 将其共性问题解决方案沉淀为应用程序框架、服务、工具等可重用基础设施, 通过 “平台化” 来避免每一个软件 “从头造轮子”. TrustieEvo 软件平台原型系统 (图 4) 即为图 1 所给 TrustieEvo 方法共性部分的物化沉淀. 作为支撑复杂关键软件系统构造和运行演化的基础设施, 它由

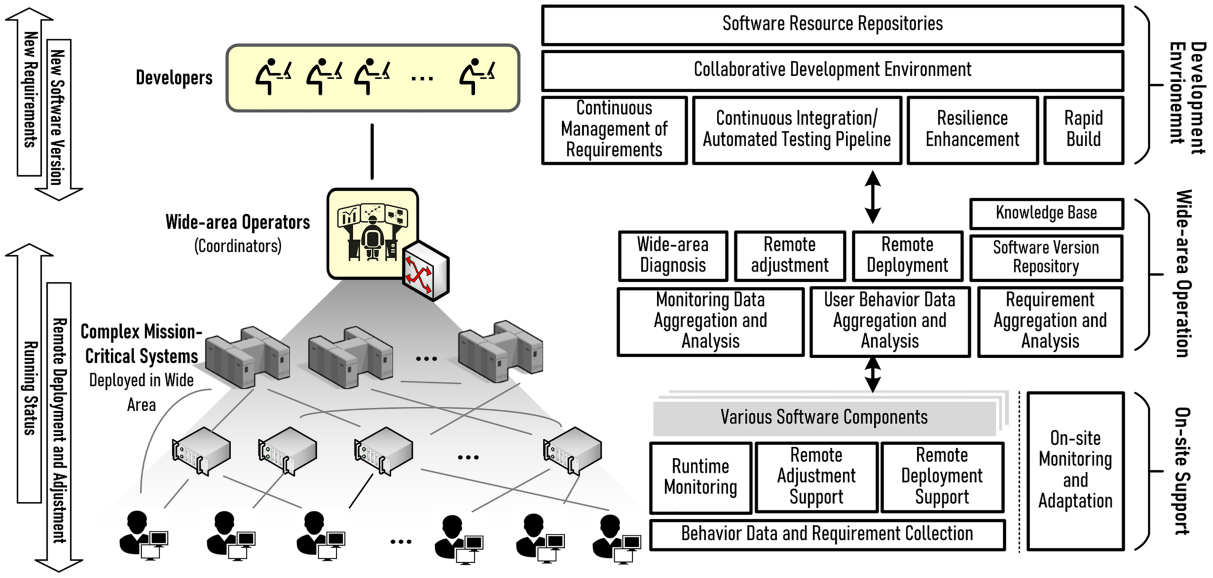

图 4　(网络版彩图) TrustieEvo 软件平台原型系统.

Figure 4　(Color online) Prototype of TrustieEvo software platform.

运行支撑、广域运控和群智开发 3 个部分组成.

运行支撑. 在运行现场直接支撑复杂关键软件系统的运行演化, 其应用程序框架、监控使能部件、演化使能部件部署于每一个计算节点, 同一现场 (如同一数据中心内部) 部署一套监控调整服务、用户行为数据和需求采集服务, 其中监控调整服务 [37] 为现场运维人员提供监控和运维入口、定制用于现场分析诊断的机器智能规则/算法.

广域运控. 为 TrustieEvo 方法的广域需求汇聚和广域开发运行一体韧性增强提供支撑, 由广域数据汇聚与分析 (含运行监控数据、用户行为数据、需求数据)、广域综合运控和推送部署、支持广域运行数据分析诊断的机器学习大模型和知识库、软件版本库等组成.

群智开发. 为复杂关键软件系统版本的快速可信迭代提供支持, 由需求持续跟踪管理、群智协同开发环境、持续集成/自动测试流水线、韧性增强、快速发布、软件资源库等组成, 其中快速发布部件包括模拟部署环境和自动部署流水线, 韧性增强部件包括在模拟部署环境中开展混沌试验的基础设施和部件.

7　TrustieEvo 案例实践

TrustieEvo 方法及平台在某复杂关键软件系统的近五年研制活动中得到应用, 本节将对实践效果及取得的经验教训进行阐述.

7.1　实践及其效果

案例3 (数据传输系统)　该软件系统运行于高度管制环境, 功能是实现广地域范围内重要数据传输. 系统架构由 “云 – 边 – 端” 体系组成, 其核心为多个数据中心 (“云”)、重要地理节点上的多个服务器集群 (“边”), “端” 则包括智能手机、平板、桌面显示器等终端设备, 以及直接嵌入到各类物理设备中的软件节点. 系统担负两类任务: (1) 日常值班运行, 支撑业务人员日常工作; (2) 突发情况处置, 在出现突发问题时, 在广地域范围内通过高可靠数据传输支撑人和物理设备的高效协同.

该系统多个旧版本的研制均严格采用瀑布模型, “需求 – 设计 – 定型发布” 循序渐进的开发周期长达数年. 近年来, 该系统部署模式从各领域独立建设向广域云边一体化过渡, 使用模式也随着领域能力生成方式的变革而不断变化. 这些变化直接表现为新的软件需求持续涌现、软件运行环境持续变化, 而传统软件工程方法以 “年”

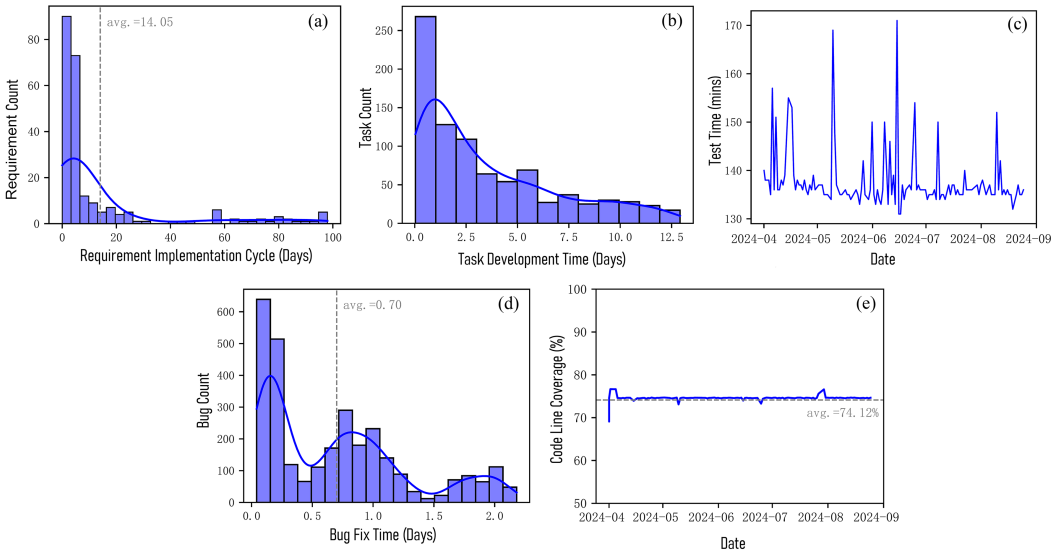

图 5　(网络版彩图) TrustieEvo 方法对于软件代谢率的提升. (a) 需求实现周期; (b) 任务开发时间; (c) 测试流水线运行时间; (d) 缺陷修复时间; (e) 自动化测试覆盖率.

Figure 5　(Color online) Software metabolism rate was improved by TrustieEvo. (a) Requirement implementation cycle; (b) task development time; (c) test pipeline running time; (d) bug fix time; (e) automated test coverage.

为单位的开发周期很难跟上用户对软件系统的预期, 成了系统建设的瓶颈性问题.

自 2020 年以来, TrustieEvo 方法成功应用于该数据传输系统新版本的研制. 在项目启动阶段, 我们将新版本初始需求纳入需求池, 基于 TrustieEvo 软件平台进行协同开发, 利用数月时间形成了被称为 “最小可用版本” 的初始版本. 与此同时, 开发人员和测试人员共同编写了数千个自动化测试用例 (截至 2024 年 12 月测试用例数达近 3 万个), 构建了每日迭代和软件正式发布自动化测试流水线, 以及用于最终验证产品能力、开展韧性试验等的模拟生产环境. 最小可用版本在上线运行以后, 项目组基于开发与运行伴随的思想, 持续对软件进行改进, 并不断发布新版本, 需求来源包括在线使用已部署版本的用户、运维部门、其他利益相关方等. 在这一过程中, 开发活动由被称为 “迭代” (iteration) 的方式进行组织, 迭代的目标是实现一组在能力维度上具有密切关联的需求项. 每个迭代会生成一个对应的内部版本, 而若干个迭代完成之后会产生被称为 “里程碑” (milestone), 满足指定需求集合的版本, 该版本将进入全量自动化测试、安全检测、韧性试验等环节, 并最终形成可推送部署版本. 换言之, “迭代” 和 “里程碑” 分别在开发活动和产品发布活动层面上推动系统的成长演化.

我们对数据传输系统某个构件从 2020 年底至 2024 年 8 月间已实现的 236 项软件需求的需求实现周期 M 进行了统计分析, 这些需求最终分解为 879 项多线并发的开发任务. 由于客观条件限制, 需求实现周期 M 中的能力形成以通过测试流水线测试、在模拟环境中形成部署能力为准. 经统计, 超过 50% 的需求可以在 7 天内实现, 超过 80% 需求的实现周期在 14 天以内, 如图 5(a) 所示. 项目的整体软件需求实现周期 (根据定义 1 计算) 由瀑布模式下的 “年” 级别降低到平均 14.05 天.

需求实现周期可进一步细分为需求纳入管理的时间、协同开发时间和测试确认时间. 由于前者与管理环节密切相关, 我们主要对相对客观的后二者进行了统计: (1) 协同开发时间. 如图 5(b) 所示, 近半数协同开发任务可在 2∼3 天内完成编码, 基本所有协同开发任务都可在两周内完成开发, 约 25% 的开发任务在一天内完成. 开发任务能够在短时间内完成, 一方面得益于基于群智范式的软件开发方法 (及其环境) 的应用, 另一方面得益于开发任务拆解粒度合理、功能逻辑清晰等因素. (2) 测试确认时间. 在应用本方法之前测试确认完全依靠手工进行, 一轮测试通常需要数周, 在应用本方法后, 在测试流水线上执行一轮完整测试时间在 130∼175 min (图 5(c)∼(e)), 自动化测试覆盖了约 74% 的代码行. 测试过程中发现的缺陷会重新进入协同开发环节, 修复时间平均少于 1 天.

我们对快速迭代过程中复用可信保证知识的混合回归测试用例选择 (参见第 5.2 小节) 的收益进行了分析, 选取了交付新版本前 (场景 1, 即该周存在版本交付压力) 及日常开发 (场景 2, 即该周不存在交付压力) 两个典

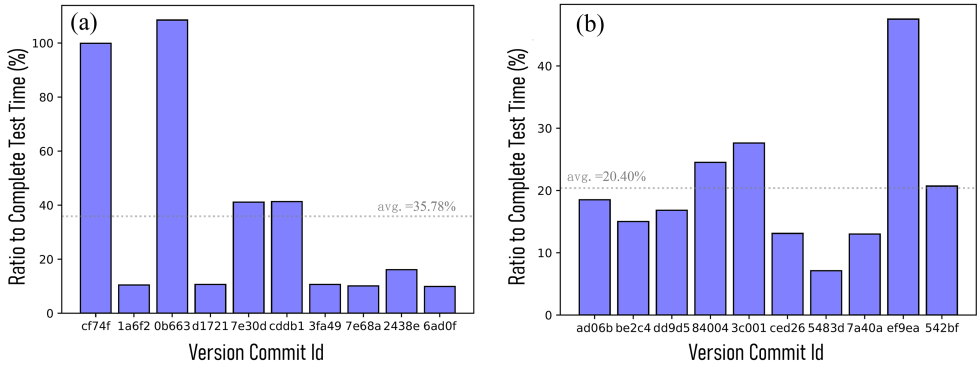

图 6　(网络版彩图) 混合回归测试用例选择对测试效率的提升. (a) 场景 1 中测试流水线运行时间; (b) 场景 2 中测试流水线运行时间.

Figure 6　(Color online) Improvements of test efficiency by hybrid regression test case selection. (a) Test pipeline running time in scenario 1; (b) test pipeline running time in scenario 2.

型场景, 场景中每天生成 1 个版本进入流水线, 每个场景选取了 10 个版本 (即 10 天) 进行数据统计. 图 6 给出了采用混合回归测试用例选择技术之后, 每个版本测试时间相对于测试流水线所有测试用例完整执行时间的百分比. 在交付/发布新版本场景下, 混合回归测试选择技术可以将测试流水线运行时间平均降低到 35.78%, 日常开发场景下则可以将测试流水线运行时间降低到 20.4%. 场景 1 和 2 的结果有差别的原因在于存在交付压力时, 压力会被传导给开发人员, 导致更多代码修改.

7.2　经验与教训

软件工程领域 “没有银弹” [25], 各种方法均有其特定的场景适用性和局限性. 在应用 TrusiteEvo 方法过程中, 观察到如下一些现象.

(1) 软件代谢率的 “长尾分布”. 图 5(a) 呈现出典型的 “长尾分布” 现象, 约 10% 的软件需求实现时间超过 56 天, 1% 的需求超过 3 个月. 经对这一部分长尾进行分析, 其产生原因包括 3 类: (i) “伪需求” 导致返工, 在应用方法时需求管理人员未对原始需求严格审核, 为了追求软件代谢率提升而直接进入开发环节, 导致实现效果不符合用户期望, 或者付出大量努力后仍无法实现, 只能将需求优先级调低; (ii) “组织孤岛” 带来的迟滞, 部分需求的实现需要跨越多个组织 (如项目组、开发团队或单位), 如果只有一个组织应用认同快速迭代思维, 而其他组织不认可这一方法, 则会明显拖慢软件代谢率, 因此在较广泛范围内建立对快速迭代演化的认同十分重要; (iii) “技术债” [38] 导致的延期, 过于追求软件代谢率, 可能会引导软件开发人员采用见效快的 “短平快” 方法, “技术债” 在累积到一定程度后可能会导致大规模重构, 解决这一问题需要依靠高层设计人员 (如架构师) 在每个微观 “开发 – 测试 – 集成 – 部署” 周期的早期及时给予指导.

(2) 软件代谢率进一步提升的潜在方法. 在现有实践中, 借助 TrustieEvo 方法和平台, 需求实现周期最短也只能达到 “天” 级别, 这是因为需求纳管、群智开发、测试确认等环节目前仍主要依赖人来驱动、人来解决复杂问题. 我们认为, 要进一步提升软件代谢率, 需要智能化手段 [39] 更多地介入, 包括人工智能辅助的需求求精、人工智能支持的群智开发和代码自动生成、人工智能驱动的软件自主适应演化、人工智能赋能的软件测试及可信保证等. 例如, 在 TrustieEvo 所强调的 “需求即数据” 原则及所构建的广域数据汇聚体系下, 海量的用户行为数据、运行监控数据被汇聚, 要准确及时地从中提取需求, 必须要有机器学习等手段的支持; 为发挥测试流水线的功效, 数据传输系统项目投入大量的人力和物力编写自动化测试用例, 也只能将测试覆盖率提升到 70% 左右, 人工智能生成软件测试用例未来可能会成为提升自动化测试覆盖率的重要手段.

(3) 制度变革、文化建设、最佳实践等的重要性. 通过对高度管制环境下多个软件开发团队的观察, 我们发现普遍的特征是对新技术的采用持怀疑态度, 原因包括担心引入风险、思维固化、很难与上级或权威管理方要求兼容、缺乏相应人才等. 因此, 开发运行一体的复杂关键软件系统演化不仅仅是技术问题, 很多时候更是管理

制度问题和团队文化问题. 另外, 我们也观察到一些团队由于在短时间内不断更换方法、工具和流程, 处于 “技术倦怠” 状态. 此时提供领域最佳实践将带来较大的参考意义.

8　相关研究工作

早期软件领域对复杂性的研究主要从工程技术角度展开, 关注结构复杂性、算法复杂性、代码复杂性、数据复杂性等 [40, 41]. 近年来, 很多软件系统越来越表现出 “系统之系统”、“信息 – 物理 – 社会” 系统、群体智能系统等系统科学意义上的 “复杂性”, 人们开始从复杂系统角度探究其开发和运行演化的规律. 钱学森在 20 世纪 90 年代初指出: “凡现在不能用还原论方法处理的, 或不宜用还原论方法处理的问题, 而要用或宜用新的科学方法处理的问题, 都是复杂性问题.” 文献 [42] 指出, 软件系统规模的变化会带来其复杂性的显著增加, 开发需要采用全新的理念、架构和方法; 文献 [43] 以分布式系统等为例, 对复杂软件系统中的涌现 (emergence) 现象进行了讨论, 并对具有负面效应的涌现现象如何分类、预测、检测、诊断和改善等研究问题进行了概述; 文献 [1] 给出了复杂软件系统的明确定义, 并给出了其成长性构造和适应性演化法则; 文献 [44] 也对复杂软件系统的概念进行了讨论; 文献 [15] 对复杂软件系统中的内部不确定性和外部不确定性进行了调研分析. 在具体机理和方法研究方面, 研究者一方面尝试使用复杂系统理论 (如复杂网络中的小世界和无标度理论等) 来理解软件系统的复杂性, 进而指导其开发运行活动 [45]; 另一方面, 针对特定某个维度上的复杂性, 对软件构造方法展开探索. 例如, 对于具有 “系统之系统” 特征的软件系统, 研究者在系统描述语言、软件验证与确认、软件系统结构、软件部署后问题、开发框架与环境等方面取得了一定成果 [46]; 针对人工群体智能系统如何构造的问题, 宏编程 (macroprogramming) 范式以及 PyoT 和 MacroLab 等框架 [47] 被提出, 可以使用编程语言在宏观尺度上定义由多个计算实体所组成系统的行为; 文献 [48] 探索了软硬件环境及外部资源不断变迁挑战下的可成长网构软件理论方法和实现技术.

任务关键软件系统的开发活动目前仍较普遍采用瀑布模型. 除第 3.2 小节给出的国防领域案例外, 医疗设备软件国际标准 ISO/IEC62304 [2]、民用航空机载软件国际标准 RTCA DO-178C [3]、轨道交通软件标准 IEC62279 [4] 等均将软件生命周期严格划分为需求、设计、实现、测试、发布等阶段, 并强调每个阶段的可追溯性. 近年来, 人们开始认识到任务关键软件系统首先是软件系统, 快速迭代对于大规模任务关键软件系统同样重要. 在迭代开发、快速形成软件版本方面, 多个研究者将敏捷软件开发方法拓展到任务关键系统中, 研究问题包括如何在迭代过程中提供合规性、如何处理需求的灵活性、如何应用测试驱动的开发方法等 [49]; 在开发与运维协同、推动软件快速进入生产环境并形成能力方面, 文献 [17] 给出了在高度管控环境中实施 DevOps 实践的挑战, 并提出了一种称为 HRE-DevOps 的评估和实施过程; DevSecOps 将软件开发运行一体化的思想拓展到安全领域, 强调开发团队、运维团队和安全团队的紧密协作, 推动软件能力的快速交付 [50]; DevOpRET 给出了开发和运行信息相结合进行持续可靠性测试的方法 [51]; 文献 [52] 从真实实践中提炼总结, 讨论了在高度管制环境下应用持续集成/持续交付流水线的问题, 并给出了若干在高度管制环境下更好实施 DevSecOps 的建议.

本文工作是文献 [1] 所给复杂软件系统成长性构造和适应性演化法则在任务关键领域的落地和深化, 给出了此类软件系统如何持续成长演化的一种可行可实施方案. TrustieEvo 方法紧密结合复杂关键软件系统服务质量可信确保、信息物理社会高度融合、利益相关方众多等特点 (参见第 3.2 小节), 通过技术框架打通此类软件系统开发与运行环节, 将各个利益相关方 (开发人员、运维人员、用户、权威管理方等) 系统化地编织到一起, 共同驱动软件快速迭代演化, 并在这一过程中针对复杂关键软件系统多实例广地域部署特点, 构建需求和运行数据持续采集、新版本持续发布的广域回路, 形成开发与运行共促的韧性增强和可信保证能力, 这是其与 DevSecOps 等其他近期工作的区别.

2) https://www.iso.org/standard/64686.html.
3) https://en.wikipedia.org/wiki/DO-178C.
4) https://webstore.iec.ch/en/publication/22781.

9　结论

复杂关键软件系统兼具复杂系统、任务关键系统和软件系统的性质, 经典按部就班、计划驱动的软件工程方法很难有效支撑其构造和演化, 使得我们在实践中不得不面临诸如 “复杂关键软件系统悖论” 的困境. 本文针对这一挑战, 提出了开发运行一体的复杂关键软件系统演化方法 TrustieEvo, 阐述了其全生命周期需求持续管理、多方协同的版本快速可信迭代、开发与运行共促的系统韧性增强 3 个方面核心使能机制, 给出了相应软件平台原型系统的设计, 并对其初步应用效果进行了分析讨论. 这一方法来自实践、指导实践, 本身也是在 “边建边用边升级”、持续迭代改进过程中, 我们后续将在方法框架下继续推进研究和实践, 下一步的主要工作方向包括如下内容. (1) 人工智能前沿技术在需求持续管理、版本可信迭代和系统韧性增强等环节中的应用; (2) 结合其他复杂关键软件系统 (如 “端” 部分为高可靠嵌入式系统的 “云 – 边 – 端” 系统、高安全要求系统等) 开展实践, 完善 TrustieEvo 方法及平台能力.

致谢　感谢数据传输系统项目组近 5 年来对本文工作的全力支持, 即使在承担繁重工程任务, 以及部分人员一开始并没完全理解新方法意义情况下, 项目组仍义无反顾地与作者共同开启了这场变革之旅. 论文成稿过程中, 王戟、张捷等多位领域专家给予了指导, 特此一并感谢.

参考文献

1 Wang H M, Ding B. Growing construction and adaptive evolution of complex software systems. Sci China Inf Sci, 2016, 59: 050101
2 Wang H M, Yin G, Xie B, et al. Research on network-based large-scale collaborative development and evolution of trustworthy software. Sci Sin Inform, 2014, 44: 1–19 [王怀民, 尹刚, 谢冰, 等. 基于网络的可信软件大规模协同开发与演化. 中国科学: 信息科学, 2014, 44: 1–19]
3 Royce W W. Managing the development of large software systems. In: Proceedings of IEEE WESCON, 1970
4 Justinia T. The UK's national programme for IT: why was it dismantled? Health Serv Manage Res, 2017, 30: 2–9
5 Defense Science Board. Design and acquisition of software for defense systems. 2018. https://www.dmi-ida.org/knowledge-base-detail/design-and-acquisition-of-software-for-defense-systems
6 Beck K. Manifesto for Agile Software Development. 2001. http://agilemanifesto.org/
7 Leite L, Rocha C, Kon F, et al. A survey of DevOps concepts and challenges. ACM Comput Surv, 2019, 52: 1–35
8 Byrne K, Cevenini A. Aligning DevOps concepts with agile models of the software development life cycle (SLDC) in pursuit of continuous regulatory compliance. In: Proceedings of Conference on Innovative Technologies in Intelligent Systems and Industrial Applications, 2022
9 Lehman M M. Programs, life cycles, and laws of software evolution. Proc IEEE, 1980, 68: 1060–1076
10 Chen J, Xin B. Key scientific problems in the autonomous cooperation of manned-unmanned systems. Sci Sin Inform, 2018, 48: 1270–1274 [陈杰, 辛斌. 有人/无人系统自主协同的关键科学问题. 中国科学: 信息科学, 2018, 48: 1270–1274]
11 Naur P, Randell B. Software Engineering: Report of a Conference Sponsored by the NATO Science Committee. NATO Technical Report. 1969. https://www.scrummanager.com/files/nato1968e.pdf
12 Snowden D J, Boone M E. A leader's framework for decision making. Harvard Business Rev, 2007, 85: 68–76
13 Szulewski P A, Maibor D S. MIL-STD-498: what's new, and some real lessons learned. In: Proceedings of Digital Avionics Systems Conference, 1995
14 Chaillan N, Yasar H. Waterfall to DevSecOps in DoD. Carnegie Mellon University Software Engineering Institute. 2019. https://apps.dtic.mil/sti/trecms/pdf/AD1085204.pdf
15 Tan C, Zhang J X, Wang T X, et al. Uncertainty-wise software engineering of complex systems: a systematic mapping study. J Softw, 2021, 32: 1926–1956 [檀超, 张静宣, 王铁鑫, 等. 复杂软件系统的不确定性. 软件学报, 2021, 32: 1926–1956]
16 Ren F Y. Internet architecture research and reflections. Acta Electron Sinica, 2024, 52: 373–384 [任丰原. 互联网体系结构的研究与思考. 电子学报, 2024, 52: 373–384]
17 Morales J A, Yasar H, Volkman A. Implementing DevOps practices in highly regulated environments. In: Proceedings of International Conference on Agile Software Development, 2018
18 Wang H M, Shi P C, Zhang Y. Jointcloud: a cross-cloud cooperation architecture for integrated internet service customization. In: Proceedings of International Conference on Distributed Computing Systems, 2017
19 CMU/SEI. Enabling the advanced battle management system vision through architecture. https://www.sei.cmu.edu/publications/annual-reviews/2020-year-in-review/year_in_review_article.cfm?customel_data- pageid_315013=315521, 2020
20 Ding B, Wang H M, Shi D X. Constructing software with self-adaptability. J Softw, 2013, 24: 1981–2000 [丁博, 王怀民, 史殿习. 构造具备自适应能力的软件. 软件学报, 2013, 24: 1981–2000]

21 Noguchi R A, Wheaton M J, Martin J N. Digital engineering strategy to enable enterprise systems engineering. In: Proceedings of INCOSE International Symposium, 2020
22 Rasheed A, Zafar B, Shehryar T, et al. Requirement engineering challenges in agile software development. Math Probl Eng, 2021, 2021: 6696695
23 Krishna M, Gaur B, Verma A, et al. Using LLMs in software requirements specifications: an empirical evaluation. In: Proceedings of International Requirements Engineering Conference, 2024
24 Wang H M, Yu Y, Wang T, et al. Crowd intelligence paradigm: a new paradigm shift in software development. Sci Sin Inform, 2023, 53: 1490–1502 [王怀民, 余跃, 王涛, 等. 群智范式: 软件开发范式的新变革. 中国科学: 信息科学, 2023, 53: 1490–1502]
25 Brooks F P. The Mythical Man-Month. Reading: Addison-Wesley, 1975
26 Wang H. Harnessing the crowd wisdom for software trustworthiness. SIGSOFT Softw Eng Notes, 2018, 43: 1–6
27 Wang T, Yin G, Yu Y, et al. Crowd-intelligence-based software development method and practices. Sci Sin Inform, 2020, 50: 318–334 [王涛, 尹刚, 余跃, 等. 基于群智的软件开发群体化方法与实践. 中国科学: 信息科学, 2020, 50: 318–334]
28 Wu Y W, Zhang Y, Wang T, et al. Towards understanding Docker build faults in practice: symptoms, root causes, and fix patterns. In: Proceedings of International Conference on the Foundations of Software Engineering (FSE), 2025
29 Wu Y W, Zhang Y, Wang T, et al. A transformer-based model for assisting Dockerfile revising. In: Proceedings of International Conference on Software Engineering, 2024
30 Wu Y W, Zhang Y, Xu K L, et al. Understanding and predicting docker build duration: an empirical study of containerized workflow of OSS projects. In: Proceedings of International Conference on Automated Software Engineering (ASE), 2022
31 Zhang G F, Liu L Y, Chen Z B, et al. Hybrid regression test selection by integrating file and method dependences. In: Proceedings of ACM/IEEE International Conference on Automated Software Engineering, 2024
32 Li M S, He M, Yang D, et al. Software cost estimation method and application. J Softw, 2007, 18: 775–795 [李明树, 何梅, 杨达, 等. 软件成本估算方法及应用. 软件学报, 2007, 18: 775–795]
33 Theunissen T, van Heesch U, Avgeriou P. A mapping study on documentation in continuous software development. Inf Software Tech, 2022, 142: 106733
34 Amiri Z, Heidari A, Navimipour N J, et al. Resilient and dependability management in distributed environments: a systematic and comprehensive literature review. Cluster Comput, 2023, 26: 1565–1600
35 Rosenthal C, Jones N. Chaos Engineering: System Resiliency in Practice. Sebastopol: O'Reilly Media, 2020
36 Ghanam Y, Maurer F, Abrahamsson P. Making the leap to a software platform strategy: issues and challenges. Inf Software Tech, 2012, 54: 968–984
37 Ding B, Wang H, Shi D, et al. CloudAmulet: promoting software reuse in datacenter application monitoring. J Software Eng, 2017, 11: 246–254
38 Digkas G, Chatzigeorgiou A, Ampatzoglou A, et al. Can clean new code reduce technical debt density? IEEE Trans Software Eng, 2020, 48: 1705–1721
39 Lo D. Trustworthy and synergistic artificial intelligence for software engineering: vision and roadmaps. In: Proceedings of IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), 2023
40 McCabe T J. A complexity measure. IEEE Trans Software Eng, 1976, SE-2: 308–320
41 Mens T. On the complexity of software systems. Computer, 2012, 45: 79–81
42 Northrop L, Feiler P, Gabriel R P, et al. Ultra-Large-Scale Systems: the Software Challenge of the Future. Carnegie Mellon University Software Engineering Institute. 2006. https://www.sei.cmu.edu/documents/ 1302/2006_014_001_635801.pdf
43 Mogul J C. Emergent (mis)behavior vs. complex software systems. ACM SIGOPS Oper Syst Rev, 2006, 40: 293–304
44 Rossi B, Pitner T. Towards a definition of complex software system. In: Proceedings of Conference On Computer Science and Intelligence Systems, 2023
45 Šubelj L, Bajec M. Software systems through complex networks science: review, analysis and applications. In: Proceedings of International Workshop on Software Mining, 2012
46 Cavalcante E, Batista T, Oquendo F. Looking back and forward: a retrospective and future directions on software engineering for systems-of-systems. J Software Evolu Process, 2024, 36: e2697
47 Casadei R. Macroprogramming: concepts, state of the art, and opportunities of macroscopic behaviour modelling. ACM Comput Surv, 2023, 55: 1–37
48 Xu C, Qin Y, Yu P, et al. Theories and techniques for growing software: paradigm and beyond. Sci Sin Inform, 2020, 50: 1595–1611 [许畅, 秦逸, 余萍, 等. 可成长软件理论方法和实现技术: 从范型到跨越. 中国科学: 信息科学, 2020, 50: 1595–1611]
49 Heeager L T, Nielsen P A. A conceptual model of agile software development in a safety-critical context: a systematic literature review. Inf Software Tech, 2018, 103: 22–39
50 Rajapakse R N, Zahedi M, Babar M A, et al. Challenges and solutions when adopting DevSecOps: a systematic review. Inf Software Tech, 2022, 141: 106700
51 Bertolino A, Angelis G D, Guerriero A, et al. DevOpRET: continuous reliability testing in DevOps. J Software Evolu Process, 2023, 35: e2298
52 Morales J A, Yasar H. Experiences with secure pipelines in highly regulated environments. In: Proceedings of International Conference on Availability, Reliability and Security, 2023

Evolution of complex mission-critical software systems based on development and runtime cooperation

Bo DING[1,2], Huaimin WANG[1,2*], Haibo MI[1,3], Zhenbang CHEN[1,2], Jun FU[1,2] & Xunhui ZHANG[1,2]

1. *State Key Laboratory of Complex & Critical Software Environment, Changsha 410073, China*
2. *College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China*
3. *Information Support Force Engineering University, Wuhan 430000, China*
* Corresponding author. E-mail: hmwang@nudt.edu.cn

Abstract Software is becoming part of the human societal infrastructure. This kind of software system is characterized by both "complexity" and "criticality". Throughout their whole lifecycle, both user requirements and running environments undergo frequent changes. Traditional software engineering approaches struggle to provide effective support for such systems, making this a pressing challenge in current practice. To meet this challenge, it is necessary to introduce the "building, using, and upgrading simultaneously" approach, allowing the system's evolution to keep pace with its external changes. However, complex mission-critical software systems operate in highly regulated environments, characterized by reliable quality of service assurance, high integration of information, physical, and social aspects, numerous stakeholders, which results in their unique software evolution laws and makes it difficult to simply copy practices from other fields, such as Internet software. Based on the analysis of the characteristics of complex mission-critical software systems, we propose an evolution method for this kind of software system, TrustieEvo, which is closely integrated with development and runtime activities and driven by multiple stakeholders. Its core enabling mechanisms are presented in three aspects: continuous requirement management, trusted version iteration, and system resilience enhancement, and the corresponding software platform design is also presented. TrustieEvo has been successfully applied to the continuous evolution of a real complex mission-critical software system for nearly five years, reducing the requirement implementation cycle from "years" to "weeks" from the proposal of new user requirements to the formation of new software capabilities, and has withstood high-intensity validation in practice.

Keywords complex mission-critical software systems, highly-regulated environment, software evolution, software metabolism rate, software resilience