

信息安全领域

AES128算法

AES128是一种对称加密算法，使用128位密钥对数据进行多轮混淆和扩散，以其卓越的性能和高度的安全性脱颖而出，成为了全球广泛应用的加密技术之一。但是AES128算法在通用处理器上面临着明显的性能瓶颈，本文采用隼瞻科技ArchitStudio™工具，为AES128算法定制了一级加速和二级加速指令扩展方案，这些指令可直接在硬件层级执行单轮加密的核心操作，将原本需要大量内存访问和逻辑运算的操作压缩为少数几个时钟周期完成，最终在少量硬件开销的前提下获得13.6倍和50倍的性能收益。



隼瞻科技
WingSemi Technology

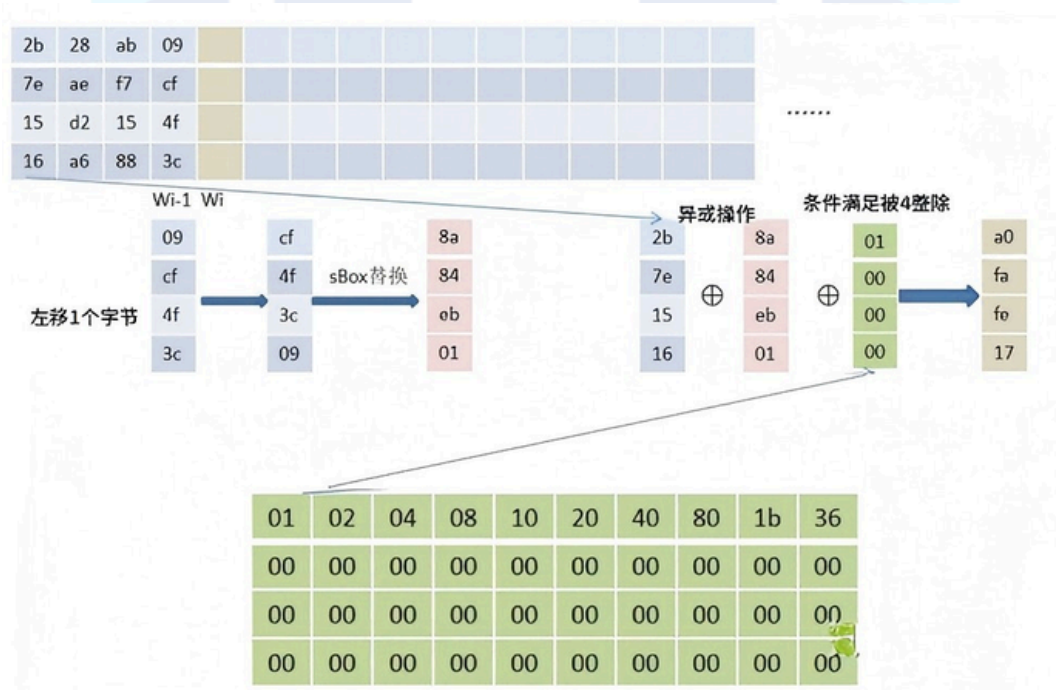
一、背景介绍

在物联网安全、数据传输加密、嵌入式设备身份认证等现代信息安全应用场景中，AES128对称分组加密算法凭借128位密钥的高安全性、轻量型的计算特性，成为终端设备端到端加密、数据存储加密的核心算法标准。AES128算法已广泛应用于智能硬件、工业控制终端、车载通信、消费电子等领域，这类场景对加密解密的实时性、硬件资源开销、功耗有着严苛要求，既需要保证加解密的处理速度满足业务流的实时性需求，又要求适配嵌入式设备的有限硬件资源与低功耗设计准则。

二、问题与挑战

1) AES128算法介绍

AES128算法以128 bit为一个明文/密文处理块，并以一个字节（8 bit）为一个单位，将数据块和密钥视为4x4字节的矩阵。算法的核心前置步骤为密钥扩展（KeyExpansion），将128位原始密钥扩展为10组128位的轮密钥，为每一轮迭代的轮密钥加操作提供密钥支持。



图：密钥拓展核心逻辑

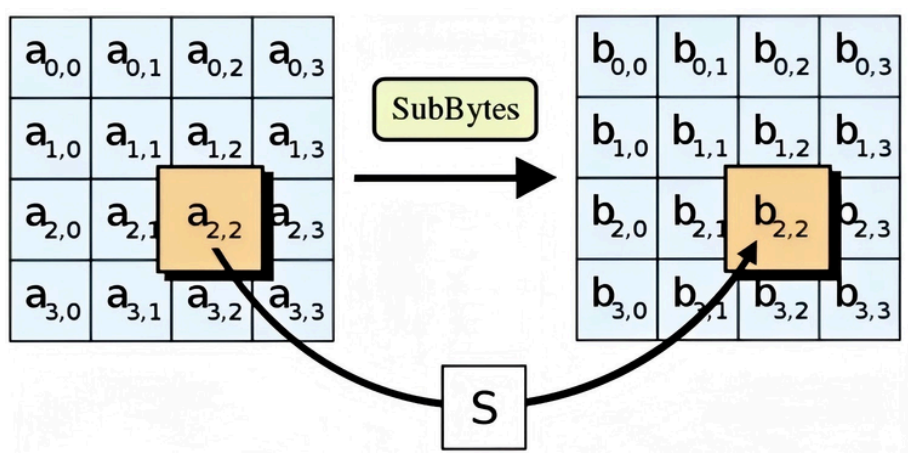
如上图所示，在密钥扩展算法中，选取初始密钥的第四列的四个字节（09，cf，4f，3c）进行左移一个字节的操作，再对左移后四个字节进行SBox查表变换。完成变换后将结果与第一列四个字节（2b，7e，15，16）进行异或操作，再与查表向量进行异或操作即可得到新一轮密钥的首列字节。最后，将新一轮密钥的首列字节与前一轮密钥的第二列字节异或则可得到第二列字节，循环往复即可得到新一轮完整密钥。对上述过程进行循环，即可将128位原始密钥扩展为10组128位的轮密钥。

```
1 void KeyExpansion(const uint8_t *key, uint8_t *round_key)
2 {
3     uint8_t temp[4];
4     int i;

5     // 初始轮密钥为输入密钥
6     for (i = 0; i < 16; i++) {
7         round_key[i] = key[i];
8     }
9
10    // 生成后续轮密钥
11    for (i = 4; i < 44; i++) {
12        temp[0] = round_key[(i - 1) * 4];
13        temp[1] = round_key[(i - 1) * 4 + 1];
14        temp[2] = round_key[(i - 1) * 4 + 2];
15        temp[3] = round_key[(i - 1) * 4 + 3];
16
17        if (i % 4 == 0) {
18            // RotWord + SubWord + Rcon
19            uint8_t tmp = temp[0];
20            temp[0] = sbox[temp[1]] ^ Rcon[i / 4];
21            temp[1] = sbox[temp[2]];
22            temp[2] = sbox[temp[3]];
23            temp[3] = sbox[tmp];
24        }
25
26        for (int j = 0; j < 4; j++) {
27            round_key[i * 4 + j] = round_key[(i - 4) * 4 + j] ^ temp[j];
28        }
29    }
30 }
```

在完成密钥扩展后，需要进行10轮迭代运算，分别是1-9轮为完整的字节代换（SubBytes）、行移位（ShiftRows）、列混淆（MixColumns）、轮密钥加（AddRoundKey）操作，第10轮省去列混淆操作；解密过程为加密的逆操作，同样进行10轮迭代，核心操作替换为逆过程，且轮密钥加为对称操作无需逆变换。

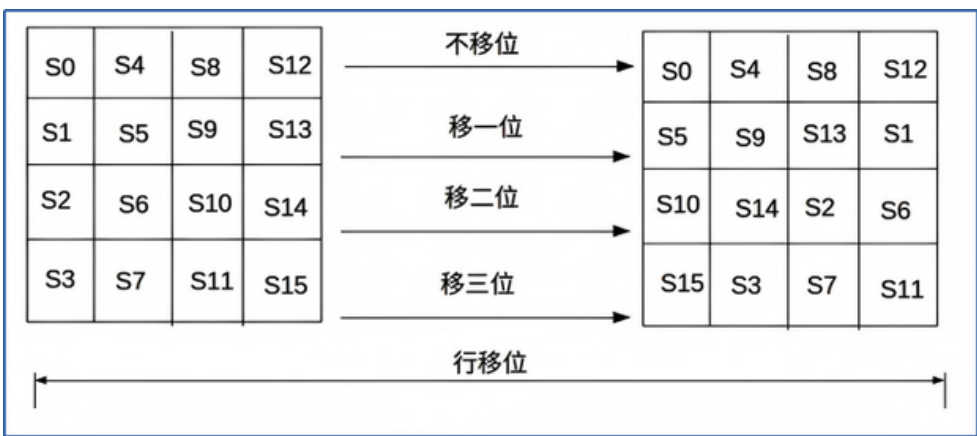
(1) **字节替换 (SubBytes)**: 将每个字节使用一个固定的查找表 (S盒, sBox) 进行替换。



图：字节替换 (SubBytes) 核心逻辑

```
1 void SubBytes(uint8_t state[16])
2 {
3     for (int i= 0; i < 16; i++)
4     {
5         state[i] = sbox[state[i]];
6     }
7 }
```

(2) **行移位 (ShiftRows)**: 行内移位操作，按特定规则将数据块的每一行向左循环移位。



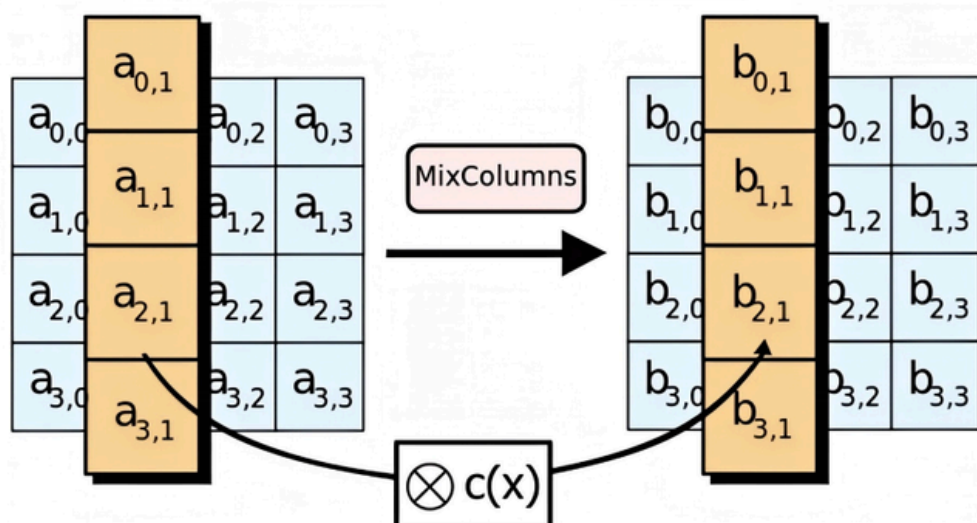
图：行移位 (ShiftRows) 核心逻辑

```

1 void ShiftRows(uint8_t state[16])
2 {
3     uint8_t tmp; //Row 1: 左移 1 字节
4     tmp = state[1];
5     state[1] = state[5];
6     state[5] = state[9];
7     state[9] = state[13];
8     state[13] = tmp;
9     //Row2:左移 2 字节
10    tmp = state[2];
11    state[2] = state[10];
12    state[10] = tmp;
13    tmp = state[6];
14    state[6] = state[14];
15    state[14] = tmp;
16    //Row3:左移 3 字节
17    tmp = state[15];
18    state[15] = state[11];
19    state[11] = state[7];
20    state[7] = state[3];
21    state[3] = tmp;
22 }

```

(3) 列混淆 (MixColumns): 列内进行线性变换, 使用矩阵乘法混合列的数据。



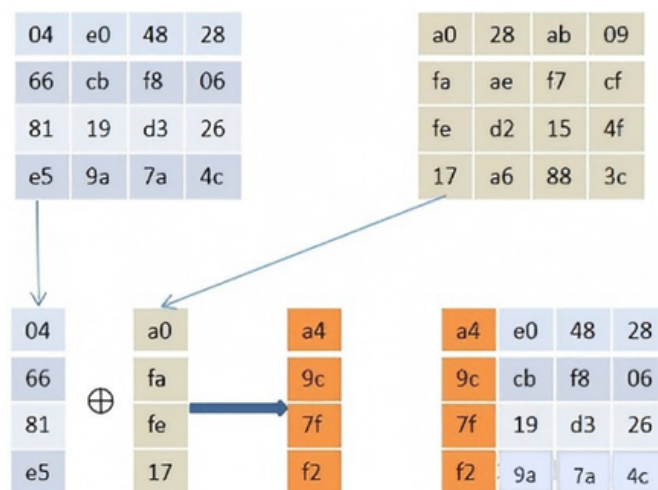
图：列混淆 (ShiftRows) 核心逻辑


```

1 void MixColumns(uint8_t state[16])
2 {
3     for (int i = 0; i < 4; i++)
4     {
5         uint8_t a = state[i * 4];
6         uint8_t b = state[i * 4 + 1];
7         uint8_t c = state[i * 4 + 2];
8         uint8_t d = state[i * 4 + 3];
9         uint8_t s0_mul2 = ((a << 1) ^ ((a & 0x80) ? 0x1b : 0x00));
10        uint8_t s1_mul2 = ((b << 1) ^ ((b & 0x80) ? 0x1b : 0x00));
11        uint8_t s2_mul2 = ((c << 1) ^ ((c & 0x80) ? 0x1b : 0x00));
12        uint8_t s3_mul2 = ((d << 1) ^ ((d & 0x80) ? 0x1b : 0x00));
13
14        uint8_t s0_mul3 = s0_mul2 ^ a;
15        uint8_t s1_mul3 = s1_mul2 ^ b;
16        uint8_t s2_mul3 = s2_mul2 ^ c;
17        uint8_t s3_mul3 = s3_mul2 ^ d;
18
19        // 列混淆，直接运算
20        state[4 * i + 0] = s0_mul2 ^ s1_mul3 ^ c ^ d;
21        state[4 * i + 1] = a ^ s1_mul2 ^ s2_mul3 ^ d;
22        state[4 * i + 2] = a ^ b ^ s2_mul2 ^ s3_mul3;
23        state[4 * i + 3] = s0_mul3 ^ b ^ c ^ s3_mul2;
24    }
25 }

```

(4) **轮密钥加 (AddRoundKey)**: 将当前的数据块与当前轮密钥进行按位异或操作。

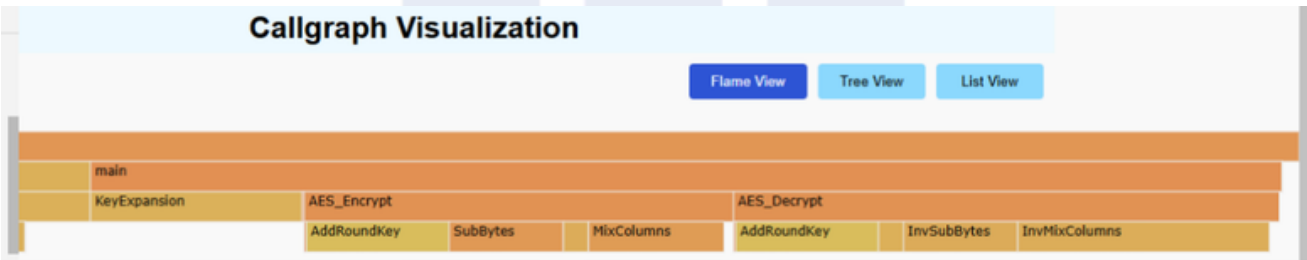


图：轮密钥加 (AddRoundKey) 核心逻辑

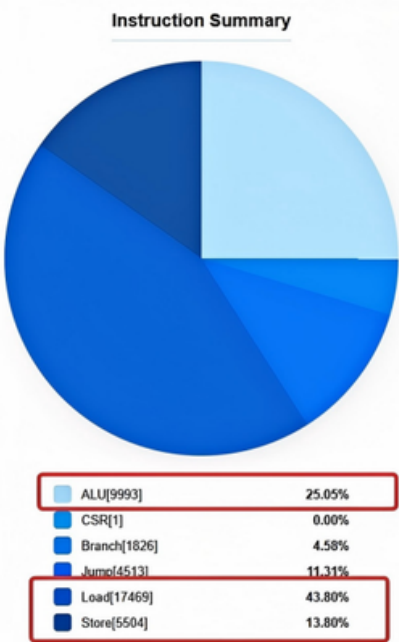
```
1 void AddRoundKey(uint8_t state[16], const uint8_t *round_key, int round)
2 {
3     for (int i= 0; i< 16; i++)
4     {
5         state[i] ^= round_key[round * 16 + i];
6     }
7 }
```

2) 通用处理器的性能瓶颈

通过隼瞻科技ArchitStudio™的Profiler性能分析工具，运行一整套AES128的加密和解密算法，可以在函数调用关系、指令统计分析、流水线分析、函数性能分析等多个维度直观展示算法在通用处理器上的计算瓶颈。



图：AES128算法热力图分析



图：AES128算法指令统计分析

如上图所示，热力图和统计分析可以直观从函数、指令等不同层次展示AES128算法面临的计算瓶颈，本案例仅展示Profiler工具部分功能，获得完整版工具即可解锁更加全面的性能分析功能。

AES128算法的硬件友好性体现在其规整的并行运算特性，但在通用RISC-V处理器上以纯C代码实现时，受限于通用指令集的运算能力，暴露出显著的性能瓶颈，主要体现在以下方面：

- **计算效率低下：**核心操作的8位数据迭代运算（如列混淆的逐字节复杂计算、行移位的逐字节移位）迭代次数较多，
- **访存延迟显著：**S盒查表、轮密钥读取、状态矩阵数据搬移涉及大量的Load/Store操作，大幅度增加处理器的功耗和性能开销。

三、解决方案

RISC-V指令集的可扩展性为解决AES128软件实现的瓶颈提供了核心途径，隼瞻科技基于ArchitStudio™工具，采用“性能分析-指令定制-硬件实现-仿真验证”的全流程方案，为AES128算法设计Scalar标量扩展指令和Vector向量扩展指令两级加速方案，将算法核心操作封装为专用硬件指令，实现“单指令/少指令完成多步运算”的硬件加速范式，同时依托ArchitStudio™的自动化能力，快速完成指令建模、RTL代码生成、工具链适配与性能验证。

1) 一级加速方案：Scalar标量扩展指令方案

Scalar标量扩展指令方案基于RISC-V基核的标量寄存器与流水线，复用基核的寄存器堆、功能单元和pipeline，针对AES128的核心操作设计专用标量扩展指令，将原有的8位数据迭代运算转为硬件实现的32位数据并行运算，减少迭代次数与指令开销，核心设计内容如下：

(1) 核心优化思路：

- **轮密钥加：**将原4个8位数据的逐字节异或迭代，转为32位数据的单周期并行异或运算，一次完成4组8位数据的异或操作。
- **列混淆/逆列混淆：**将列混淆中8位数据的异或、与、移位、条件等复杂的组合操作，固化为硬件逻辑，通过eMixColumns指令实现32位数据的并行列混淆运算，省去软件迭代的指令开销。
- **行移位/逆行移位：**将行移位中大量的Load/Store数据搬移操作，通过eShiftRows指令实现硬件级的状态矩阵位域置换/逻辑移位，彻底消除软件实现的访存指令开销。
- **字节代换/逆字节代换：**将S盒/逆S盒查表操作固化为硬件查表逻辑，通过eSubBytes指令实现多字节的并行查表替换，避免软件查表的频繁访存。

(2) 硬件实现特性：

- 标量扩展指令通过扩展指令译码接入RISC-V基核流水线，复用基核的ALU与操作数前递网络，新增专用功能单元实现算法核心运算，无额外的寄存器堆与流水线开销，硬件资源占用少。

2) 二级加速方案：Vector向量扩展指令方案

Scalar标量扩展指令方案实现了算法性能的大幅提升，但仍受限于基核32位标量寄存器与原有数据通路，存在部分Load/Store与分支指令开销。为进一步突破性能瓶颈，设计**Vector向量扩展指令方案**，构建**128位宽的专用数据通路、向量寄存器堆与向量TCM（紧耦合存储器）**，适配AES128 128位状态矩阵的原生运算需求，实现全流程的128位数据并行运算。

(1) 向量扩展指令设计：

- 针对128位并行运算需求，设计vAddRoundKey、vMixColumns/vInvMixColumns、vShiftRows/vInvShiftRows、vSubBytes/vInvSubBytes向量扩展指令，一次指令即可完成128位状态矩阵的完整核心操作，实现“单指令完成一轮运算”的极致加速。

(2) 专用硬件微架构设计：

- **16×128bit向量寄存器堆：**设计专用于AES128的向量寄存器ev[0]-ev[15]，原生存储128位状态矩阵与轮密钥，消除数据搬移的访存开销。
- **向量TCM：**定义基地址为0x39000000、大小为0x10000的向量紧耦合存储器，为向量运算提供高带宽、低延迟的存储支持。
- **独立向量流水线：**构建DE-VEX-VWB独立向量流水线，与RISC-V基核流水线解耦，实现向量指令的并行执行，核心运算在向量ALU（evALU）中完成。
- **专用功能单元：**新增 SubBytes、ShiftRows、MixColumns、AddRoundKey及其逆操作的专用硬件模块，实现128位数据的全并行运算。

四、实现效果

基于隼瞻科技M130处理器基核，分别实现纯软件C代码、Scalar标量扩展指令、Vector向量扩展指令三种AES128算法实现方案，通过ArchitStudio™的仿真器与Profiler工具完成指令数、执行周期、硬件资源的量化测试，核心性能与资源指标如下：

指标	基核C实现	Scalar扩展方案	Vector扩展方案
Cycles	32031	2344	604
收益幅度	1倍	13.6倍	50倍
面积开销（归一化）	+0% Gates	+2-3% Gates	+40% Gates

图1：性能优化结果对比

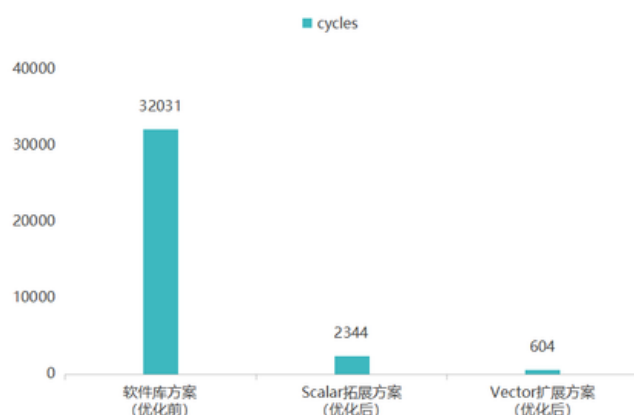
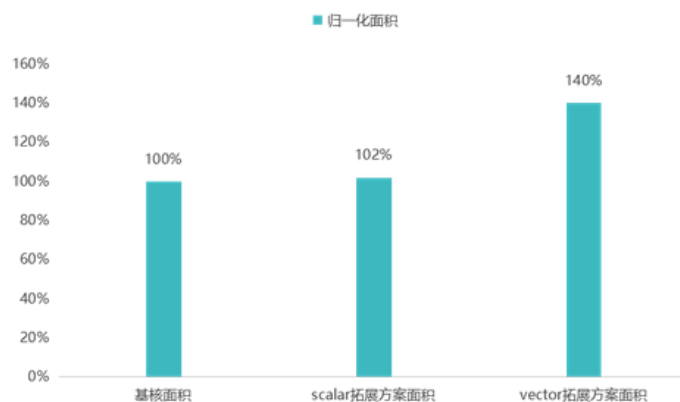


图2：面积开销结果对比



在该案例中，通过为RISC-V处理器添加标量拓展指令，复用基核的寄存器堆、功能单元和pipeline，仅增加2-3%的硬件面积开销，就为AES128算法获得13.6倍的性能提升。为了进一步提升性能，通过为处理器设计向量流水线、寄存器和vTCM，进一步提升了计算的并行性，并减少了访存的指令开支，在增加40%的硬件面积的前提下，获得50倍的性能收益，由于向量寄存器和vTCM可以被其他算法的向量拓展指令复用，因此性能收益和面积开销的比例实际上将更高。

在实际应用中，芯片架构团队通常会将算法进行软硬件划分，将算法和模型划分为软件或硬件执行。软件执行的算法具有控制和调度密集的特点，而硬件执行的算法有计算和访存密集的特点。工程师可采用ArchitStudio™对算法的热点函数进行性能分析，快速的探索不同软硬件划分方案的性能收益和面积开销，灵活的为RISC-V处理器拓展自定义指令，并设计配套的硬件加速器，最终实现以最少的面积开销收获最佳的性能提升和功耗优化，形成垂类领域最优的计算子系统方案，ArchitStudio™的智能化敏捷开发能力为上述架构快速探索提供了可能性。