ChatGPT

Windows平台C/S仿真管理系统方案设计 (ANSYS 2022R2: Fluent/LS-DYNA/MAPDL)

本方案文档详细描述了在 Windows 平台下构建一套 C/S 架构的仿真管理系统的设计与实现。系统基于 FastAPI 提供 Web 服务前端，支持用户上传 ANSYS 2022 R2 所支持的仿真任务脚本（包括 LS-DYNA 的 `.k` 文件、Fluent 的 `.cas/.dat` 或 `.cas.h5` 文件、Mechanical APDL 的 `.inp/.mac/.cdb` 文件），并通过后端工作节点或调度系统执行相应仿真计算。方案涵盖了任务上传与类型识别、参数与许可校验、仿真调度执行、结果后处理与多结果集导出、以及两种不同实现方式（直接Python API调用 vs 批处理命令）的比较分析，并提供模块化的生产级代码示例。

系统架构与流程概述

系统采用典型的**客户端-服务器（C/S）架构**，用户通过 Web 前端界面与系统交互。后端包含 FastAPI 应用和若干仿真执行节点，整体架构与工作流程如下：

- **Web客户端**：提供用户友好的界面，用于上传仿真任务文件、配置仿真参数、监视任务状态并下载结果。
- **FastAPI服务器**：作为中枢，接收用户请求（如任务上传、状态查询、结果下载），将仿真任务分发给后端执行模块。FastAPI 提供REST API接口以及WebSocket或轮询用于实时状态更新。
- **任务调度模块**：负责将上传的任务根据类型分发到对应的仿真执行模块（Fluent、LS-DYNA、MAPDL）。可以采用**异步任务队列**（如 Celery）或后台线程的方式执行仿真，使得长时间运行的仿真不会阻塞主线程。任务调度模块也负责管理任务队列和任务状态（如队列中、运行中、已完成、失败等）。
- **仿真执行模块**：针对不同仿真软件（Fluent、LS-DYNA、MAPDL）封装具体的执行方法。每种仿真类型实现统一接口，如 `run()` 执行仿真、`post_process()` 进行后处理、`check_success()` 判断成功与否等。执行模块可以通过两种方式调用仿真软件：（1）**Python API接口**调用仿真引擎，（2）**调用外部可执行程序**运行仿真批处理。稍后将详细比较这两种方案。
- **后处理模块**：在仿真完成后自动执行结果处理，包括将结果提取为 **CSV**（数值结果）和 **PNG**（图形可视化）格式。如某些仿真无法直接导出CSV/PNG，则可采用 **VTK**、**JSON** 或 **Excel** 等替代格式，确保结果可被用户下载和进一步分析。后处理模块支持**多个结果集**的识别（例如瞬态仿真的多个时间步、或结构仿真的多个工况），用户可在Web界面选择需要导出的特定结果集。
- **配置与许可管理**：系统使用集中配置文件管理所有仿真可执行文件路径、命令模板、默认参数及许可信息。运行任务前，会根据配置检查**用户指定参数是否合法**，例如CPU核数是否超过许可支持的范围，如不支持则在提交阶段拦截任务并提示用户。

用户典型操作流程为：上传仿真文件 → 选择/修改仿真参数（可从默认模板加载）→ 提交任务 → 后端分配执行 → 用户查看实时状态（排队/运行/完成/错误）→ 仿真完成后选择需要的结果集导出CSV/PNG。整个流程中的关键技术点和实现细节将在下文各节中详细阐述。

仿真任务上传与类型识别

1. 支持单文件和多文件任务上传： 仿真任务可能由单个文件或多个文件组成。例如，LS-DYNA 通常用单一的 `.k` 格式输入文件；Fluent 则可能由 `.cas` 和 `.dat`（或合并后的 `.cas.h5`）文件对组成；Mechanical APDL

(MAPDL) 可能需要 `.cdb` 几何、`.inp` 载荷工况、甚至 `.mac` 宏文件等。为支持多文件场景，前端可提供**压缩包上传**（如zip）或多文件一起上传的功能。后端接收后，如是压缩包则解压，或者根据文件扩展名自动配对，例如当检测到 `.cas` 文件同时存在 `.dat` 文件，则认为这是一个Fluent双文件任务包。

2. 自动判断任务类型： 系统根据**文件扩展名和内容**识别仿真类型：
- 扩展名 `.k` 或 `.key` 对应 **LS-DYNA**（关键字输入文件）。
- 扩展名 `.cas` / `.dat`（或 `.cas.h5`）对应 **Fluent**（流体仿真案例和数据文件）。
- 扩展名 `.cdb`、`.inp`、`.mac`、`.dat`（注意 MAPDL 也可能用 `.dat` 作为输入）等，则属于 **ANSYS MAPDL**（Mechanical APDL）。

如果一个压缩包内包含上述组合，比如同时有 `.cas` 和 `.dat`，则识别为Fluent任务；如果有 `.inp` 而无 `.cas` / `.k` 则识别为MAPDL任务，等等。实现上，可以维护一个**文件后缀到仿真类型**的映射，并根据上传文件集匹配规则来判断。如果识别冲突或不确定（例如同时存在Fluent和LS-DYNA文件），则可以根据用户在前端选择的仿真类型为准，或者要求用户明确选择。

一旦识别出类型，FastAPI服务器就会将任务封装为相应类型的仿真任务对象，并交由调度模块进入等待队列或直接由空闲工作节点执行。

仿真参数模板与许可校验

3. 仿真参数配置： 每种仿真类型通常都有可配置的参数，例如：并行计算的CPU核数、迭代步数或收敛准则、模拟总时间或步长等。系统应提供**默认参数模板**（可定义在配置文件中），用户上传任务后可自动加载该任务类型的默认参数值，同时允许用户在Web界面进行修改。这样既保证常规情况下用户无需每次填写大量参数，又给予高级用户调整参数的灵活性。

4. 许可支持校验： 仿真任务在正式排队执行前，需要检查所需的许可证是否充足，尤其是**并行计算核数**和**求解模块许可证**。例如：
- **Fluent并行**：Fluent标准许可通常包含4核并行能力，额外核心需要 HPC 包许可证支持。根据Ansys HPC Pack授权规则：**1个HPC Pack许可允许最多8核并行，2个Pack允许32核，并以此类推** [1]。例如，如果只有1个HPC Pack，则用户请求超过8核应被拒绝；有2个Pack则最多32核 [1]。系统可以在配置文件中定义各仿真类型当前可用的HPC Pack数量或直接可用核数上限，并在任务提交时验证用户输入。
- **LS-DYNA并行**：LS-DYNA许可通常分为共享内存并行（SMP）和分布式并行（MPP）两类，许可可能限制最大核数或进程数。应根据许可证配置设置LS-DYNA允许的 `ncpu` 最大值。如用户请求超出则拦截。LS-DYNA自带的 `lstc_qrun` 等工具可以查询当前许可可用核数 [2] [3]，不过通常静态配置上限即可。
- **MAPDL并行**：Mechanical APDL（ANSYS Classic）并行通常受 HPC许可限制，例如无HPC则默认2核（1核+1辅助），每个HPC Pack扩展倍数。配置文件可设定 MAPDL 的 `-np` 最大值，根据当前许可情况调整。
- **功能模块许可**：例如 LS-DYNA 算法在ANSYS中属于显式动力学模块，需要“ANSYS LS-DYNA”许可证；Fluent需要“CFD”许可等。如果用户提交的求解器在当前许可配置下不可用，系统也应立即给出错误而非排队执行。

许可校验实现上，可在FastAPI接收到提交请求时进行：读取配置中许可限制，与用户参数比对。不通过则返回错误给前端。这样做能避免占用计算资源运行后才失败，也提高了用户体验。

集中配置管理

5. 配置文件统一管理： 系统使用集中**配置文件**（如 YAML/JSON 或 Python模块）管理所有关键路径和命令模板，包括： - **可执行文件路径**：例如 `fluent.exe`、`ls-dyna.exe`、`ansys2022r2.exe` 的安装路径或调用命令。由于不同环境路径可能不同，集中配置便于运维修改，而不需改代码。
- **命令模板**：每种仿真类型的运行命令格式。例如：Fluent 命令模板可定义为：`"{solver_path} {dim} {precision} -g -t{cores} -i {journal} -o {output}"`，由代码填入实际值（如3ddp表示3D双精度，cores为核数等）；LS-DYNA 命令模板如：`"{solver_path} i={kfile} memory={mem}m ncpu={cores} o={output}"`；MAPDL 命令模板如：`"{solver_path} -b -p ansys -np {cores} -i {input} -o {output}"` 等。集中模板方便根据软件版本调整参数或增加调试标志。
- **默认参数**：各仿真类型的默认并行核数、默认迭代次数、收敛判据等默认设置。也可以包含不同分析类型的模板（如Fluent稳态VS瞬态的不同默认迭代步等）。
- **文件路径**：例如仿真临时工作目录、结果输出目录、日志目录等。所有路径统一由配置提供，在代码中引用，便于部署时按环境修改（如Windows上的盘符、集群共享存储路径等）。
- **许可设置**：如上文提到的许可核数上限、HPC Pack数量、License Server地址等。

通过集中配置，运维人员可以**一处修改**即影响全部相关模块。例如更换ANSYS版本时，只需更新路径和命令模板中版本号，新任务即可使用新版本运行。下面给出一个示例配置（YAML格式）的片段：

```
simulation:
  fluent:
    solver_path: "C:/Program Files/ANSYS Inc/v2022R2/fluent/bin/fluent.exe"
    run_cmd_template: "{solver_path} {dim} -g -t{cores} -i {journal} -o
{output}"
    default:
      dim: "3ddp"        # 3d double precision
      cores: 4
      iterations: 100   # default iterations or time steps
    license:
      max_cores: 8      # Max cores allowed (e.g., based on 1 HPC Pack = 8
cores) 1
  lsdyna:
    solver_path: "C:/Program Files/ANSYS Inc/Shared Files/LSDyna/ls-dyna.exe"
    run_cmd_template: "{solver_path} i={kfile} ncpu={cores} memory={memory}m"
    default:
      cores: 4
      memory: 4000      # in MB
    license:
      max_cores: 16     # e.g., some configured limit
  mapdl:
    solver_path: "C:/Program Files/ANSYS Inc/v2022R2/ansys/bin/win64/ANSYS.exe"
    run_cmd_template: "{solver_path} -b -p ane3fl -np {cores} -i {input} -o
{output}"
    default:
      cores: 2
      product: "ane3fl"  # e.g., LS-DYNA explicit product license in MAPDL if
```

```
needed
    license:
      max_cores: 4
global:
  work_dir: "D:/SimTasks"    # base directory for storing task files and results
  ...
```

以上配置示例定义了各仿真器的可执行路径和命令格式、默认参数和许可限制等。代码中将读取该配置用于后续操作。

仿真执行方式：Python API vs 批处理命令

系统支持两种仿真任务执行方案：(A) **通过官方Python API接口调用控制仿真流程和后处理**，或 (B) **通过命令行批处理方式运行仿真和后处理**。下面将分别介绍两种方案，并针对 Fluent、LS-DYNA、MAPDL 三类仿真器分析其适用性、优缺点和推荐落地实现。

方案A: Python API 控制仿真

概述： 通过官方提供的 Python 接口直接驱动仿真软件内核，控制从前处理、求解到后处理的流程。这种方式下，Python 脚本相当于客户端，通过API向仿真软件发送指令，获取数据。ANSYS 2022R2 开始逐步提供了更多Python接口：

- **Fluent – PyFluent**：ANSYS 在2022R2推出了 **PyFluent** Python接口库，允许开发者以Python脚本方式访问Fluent的全部功能，包括前处理（如生成/导入网格），设置物理模型，启动求解以及后处理 [4] [5]。PyFluent 实际由多个组件构成：`ansys-fluent-core`（核心求解与设置接口）、`ansys-fluent-parametric`（参数化流程接口）、`ansys-fluent-visualization`（与PyVista和Matplotlib集成的后处理可视化接口） [6]。开发者可以像使用 NumPy 等库一样，在Python中调用Fluent功能。这种模式下，Fluent求解器通常仍在本地启动，但通过gRPC或本地进程通信接受 Python 指令。 **后处理能力**：PyFluent 提供对场数据的访问，可提取计算结果数据数组，并借助 `ansys-fluent-visualization` 使用 PyVista/Matplotlib 渲染等高质量图形 [7]。例如，可以编写Python代码获取某截面的流场数据输出为CSV，或直接生成云图PNG。

- **Mechanical APDL – PyMAPDL**：ANSYS 提供了 **PyMAPDL** (ansys-mapdl-core) 库，通过 gRPC 方式远程控制 MAPDL 会话 [8]。开发者可以启动一个本地或远程的 MAPDL 实例，然后使用 Python 调用其命令。PyMAPDL 提供面向对象的后处理接口，如 `mapdl.post_processing` 类，可以直接查询节点应力位移等数组，甚至直接绘图 [9]。例如，`mapdl.post_processing.nodal_displacement("X")` 会返回所有节点的X向位移数组 [10]；也可以调用 `plot_nodal_displacement("X")` 自动生成位移云图。 [10]。PyMAPDL 同样支持仅使用结果文件进行后处理（无需在同一进程完成求解），因为其底层也可调用 DPF。另外，在2022R2时代，ANSYS正推广 **DPF (Data Processing Framework)** 框架，PyMAPDL文档中也建议对大型结果文件优先使用DPF进行后处理 [11]。**后处理能力**：借助 PyMAPDL，用户可以无需手动编写APDL宏，通过Python直接读取结果并生成所需数据/图表，例如将 `mapdl.post_processing.nodal_displacement("ALL")` 转成 NumPy 数组或 Pandas DataFrame，然后保存为 CSV。这提供了比手工解析 RST 文件更便捷的方式。PyMAPDL 也能检查仿真是否成功完成（例如通过捕获 MAPDL 提示或读取返回的字符串中是否包含 “RUN COMPLETED” 等）。

- **LS-DYNA**：LS-DYNA 相对而言 **缺乏直接的Python控制API**（LS-DYNA本身不是ANSYS原创软件）。目前ANSYS没有类似PyDyna库。但在后处理方面，ANSYS DPF 框架已经开始支持LS-DYNA的结果文件读取：**DPF支持读取 LS-DYNA 二进制结果**（如 `.d3plot`, `.binout`） [12]。因此，虽然对LS-DYNA的求解流程无法通过ANSYS提供的Python库直接控制（需要通过输入文件），但**后处理可以借助DPF**。另外，LS-DYNA官方有一个前后处理可视化工具 LS-PrePost，支持通过内部命令语言或脚本生成图片和导出数据。不过LS-PrePost的脚本体系较为独立，这里更倾向利用 DPF 或 ANSYS Mechanical 通用工具。**后处理能力**：DPF 可以将 LS-DYNA 的 d3plot 结果加载为结果对象，然后提取应力应变等场变量，并支持一些基本计算和绘图 [13]。这使我们可以用统一的Python框架对LS-DYNA结果进行处理，如获取某节点的加速度时程、输出为CSV，或者用PyVista渲染变形图。PyDPF-Post的官方文档就指出：最新DPF版本已支持 Mechanical APDL、LS-DYNA 和 Fluent 等求解器的结果文件 [12]。

方案A优点：

- 深度自动化与集成：通过API可以细粒度控制仿真过程，甚至可以动态调整流程。例如可以在Python中先调用网格划分，再设置边界条件，然后启动求解，监控收敛曲线，自动判断收敛后提前终止，最后直接提取关键结果。这些逻辑用批处理脚本难以实现，但用Python编程相对容易。
- 统一的后处理流程：使用Python接口可以将不同仿真结果都以Python对象形式获取，然后使用通用的库（如NumPy/Pandas/Matplotlib）进行处理，减少手工格式转换。例如可通过 PyDPF 统一读取不同求解器结果 [12]，将其转为数组后存CSV或画图。这提高了不同仿真类型间结果处理的一致性。
- 错误捕获与判断方便：通过API调用，可以直接获得调用是否成功的返回。例如 PyMAPDL 执行命令会有返回字符串或错误异常抛出，可据此判断是否出错；PyFluent 同样可以捕获Fluent内部错误。如果使用批处理，则需要解析日志文本来判断（相对麻烦）。
- 丰富的可编程性：Python接口允许灵活编程，例如结合第三方库（SciPy、TensorFlow等）实现优化、数据分析等扩展功能 [14]。这一点是批处理方式很难做到的。

方案A缺点：

- 学习曲线和开发成本：开发人员需要熟悉各仿真器的Python接口库的用法（例如PyFluent、PyMAPDL的API），并处理这些库可能带来的配置复杂性（例如需要确保Python环境和ANSYS版本匹配，配置ANSYS产品路径等）。
- 运行开销：通过API调用，有时仍需后台启动仿真软件实例（如PyFluent会启动Fluent进程）。与直接命令行相比，多了一层通信，可能在启动和大数据传输时稍有开销。不过PyFluent/DPF等采用高效通信机制，开销可以接受。
- 新技术稳定性：PyFluent等在2022R2刚推出，还在快速迭代，某些功能可能不够成熟或有Bug，需要关注版本兼容性。而命令行方式经过多年使用相对成熟稳定。
- *LS-DYNA*支持有限：正如前述，LS-DYNA缺少等价的Python求解控制接口，因此对于LS-DYNA求解部分仍可能需要借助方案B来完成。方案A对LS-DYNA主要能用于结果处理阶段（DPF），无法控制求解启动。

方案A适用性总结：

- *Fluent*：高度推荐使用PyFluent接口来实现自动化流程 [4]。Fluent原有Scheme语言宏或TUI命令虽也可用，但Python接口更现代友好，且和其它工具集成方便。通过PyFluent，可以完全控制Fluent的网格、设置、运行、结果输出，并直接在Python中获取结果数据或利用PyVista绘图，非常适合我们这种管理系统自动输出CSV/PNG的需求。
- *MAPDL*：推荐使用PyMAPDL或DPF接口。对于需要迭代交互控制的情况，PyMAPDL可以直接发送APDL命令、监视进程；对于单批次求解，直接调用Mapdl.run也可等待完成。结果读取则PyMAPDL和DPF均可胜任。尤其DPF可以避免传输巨大的RST文件，只提取所需数据 [8]。
- *LS-DYNA*：可部分采用方案A。虽无法用API启动LS-DYNA求解，但可以在后处理阶段通过DPF来读取d3plot数据，进而统一输出结果。因此建议**组合策略**：LS-DYNA求解本身用方案B（命令行启动），完成后在Python中用DPF加载结果以输出报告和图表。这样用户体验一致，又充分利用了ANSYS官方数据接口。

方案B: 命令行批处理执行

概述： 通过构建命令行调用字符串，直接在操作系统中启动仿真软件的批处理模式，执行完成后通过输出文件和日志判断结果、进行解析和后处理。所有三种仿真程序均支持在**无图形模式/批处理模式**下运行，给定输入文件和参数即可完成仿真计算。

- **Fluent批处理**：Fluent 提供命令行接口，可用开关指定并行核数和执行的Journal文件。例如常见调用：`fluent 3ddp -g -t4 -i job.jou -o output.log` [15]。其中`3ddp`表示3D双精度版本，`-g`禁用图形界面，`-t4`使用4核并行，`-i job.jou`执行指定的Journal命令文件，并将屏幕输出重定向到`output.log` [16] [15]。Journal文件是一系列 Fluent TUI 或 Scheme 指令的集合，可涵盖从读取case/dat、设置初始条件、运行计算、到保存结果/导出数据的所有步骤 [17] [16]。也可以通过在Fluent中使用"**File→Write→Start Journal**"录制GUI操作来生成，但更推荐手工编写TUI指令确保兼容性 [18]。本系统中，可在任务执行时自动生成或填充一个Journal模板，例如将用户上传的`.cas/.dat`名称写入`/file/read-case-data`命令，在计算完毕后包含`/report/output...`或`/file/export...`命令将结果导出为CSV，再`exit yes`退出 Fluent。**后处理**：Fluent 批处理中就可以直接完成部分后处理，如使用`/plot`或`/report`命令导出收敛曲线数据、场变量数据等至文件。Fluent还能通过TUI保存图片（`/display/set-window`配置视图，然后`/display/save-picture`保存PNG），但在无GUI模式下保存图片有时需要确保环境支持渲染（Windows下可行，Linux下需虚拟帧缓冲）。如果Fluent本身难以直接截屏保存，则可以考虑让其导出Ensight Case Gold格式，然后由**Ensight**或**PyVista**另行生成图片 [19]。不过对于大多数数值报告，Fluent的文本导出已足够。

- **LS-DYNA批处理**：LS-DYNA 典型运行方式就是命令行启动求解。例如调用可执行程序并指明输入关键字文件：`lsdyna.exe i=task.k memory=4000m ncpu=8`。LS-DYNA在Windows上可能有多个可执行版本（MPI并行、SMP并行等），这里选择合适的并行模式。命令参数中，`i=`指定输入文件，`memory=`设置内存大小，`ncpu=`设置并行线程数（SMP模式），MPI模式下可能用`np=`或通过环境MPI启动。可以通过将命令和参数写入一个`.bat`批处理文件由系统调用。**后处理**：LS-DYNA求解完成后，会生成多种输出文件：包括**文本日志**（一般是`<jobname>.mesg`或`<jobname>.out`，里面含模型信息、步骤进度和错误警告等）、**状态文件**（`.stat`，更新当前步信息）、**结果文件**（`.d3plot`二进制结果数据库，此外根据*DATABASE*卡可能有`nodout`, `elenct`, `glstat`等 *ASCII*文件）。由于 *LS-DYNA* 本身不具备直接导出 *Excel/CSV* 的功能（除了那些 *ASCII*文件已经是*CSV*格式），我们需要解析这些输出。例如可以 读取`nodout`（若用户在输入中请求输出节点数据）来提取某节点的时程数据并存为 *CSV*。图片方面，可采用*LS-PrePost*的脚本接口：编写一个*LSPP*脚本，在其中打开*d3plot*，生成等值云图并保存图片。这可通过命令行调用`lsprepost.exe c=script.cfile -nographics`来批量执行。但由于*LS-PrePost*脚本较为晦涩，本方案优先考虑利用*DPF**读取d3plot然后用Python绘图。DPF从LS-DYNA结果提取数据的能力在上节已述 [13]。因此，方案B下LS-DYNA的后处理可以是：运行结束后，调用Python脚本加载d3plot（通过DPF）并将关键结果写CSV/绘图。

- **MAPDL批处理**：Mechanical APDL 长期以来支持命令行批处理模式，用于集群或后台计算。典型调用：`ansys.exe -b -p <license> -np 4 -i input.inp -o output.out`。其中`-b`表示batch模式，`-p`指定所用许可产品（如`ane3fl`表示显式动力学LS-DYNA模块，或`ansys`表示标准机械许可），`-np 4`要求使用4核并行，如果有并行能力；`-i`指定输入文本文件（包括APDL命令流），`-o`指定输出日志文件。用户上传的`.inp`可以是 Mechanical APDL 的命令流，里面可以包含前处理网格（若`.cdb`提供模型则inp里用`/INPUT`读入）、载荷施加、求解(`SOLVE`)，以及后处理指令。我们可以在`.inp`尾部加入APDL后处理命令，比如`PRNSOL`输出结果或`/GRAPHICS`绘图保存。如果用户未提供，我们也可准备一个**通用后处理宏**：根据仿真类型自动插入。例如结构静力问题，可以在求解后加入：`/`

`POST1; SET,last; PRNSOL,U,X,; PRNSOL,USUM; ETABLE,Sigma, S,EQV; PRETAB,Sigma` 等命令将位移和等效应力打印，ANSYS会在 `output.out` 文本中列出结果，可解析提取[20]。也可以用 `*EXPORT` 命令将表格导出CSV文件（ANSYS 18.0起支持将结果写CSV）。对于图片，MAPDL可以用 `/SHOW,PNG` 打开PNG图形驱动，随后使用 `/DSCALE,/AUTO,1` 等调整，然后 `PLNSOL` 绘制云图，再 `/WRITE` 保存图片为PNG文件。当以 `-b` 无图形模式运行时，需要配置虚拟图形设备（ANSYS提供 `/SHOW,VR,` 虚拟设备支持图像输出）。当然，如果以上方法复杂，依然可以借助DPF来获取MAPDL结果然后绘图：因为MAPDL求解会生成 `.rst` 文件，**DPF天然支持ANSYS Mechanical的.rst结果**[12]。所以方案B另一种做法是：批处理完成后，由Python脚本用DPF读取生成CSV和PNG。

方案B优点：

- 实现简单成熟：批处理方式等同于平时手动运行软件，相当可靠。无需了解内部API，只要写好命令和输入文件即可。对开发人员要求较低（只需熟悉基本命令行参数和脚本语法）。很多现有流程（尤其在传统HPC场景）就是批处理模式，容易迁移。
- 与软件解耦：通过外部进程调用，主程序只是等待其结束，仿真软件崩溃也不会直接导致管理系统崩溃，只需捕获错误码或超时即可。稳定性上有所保障。
- 求解速度无额外损耗：相比API需要通过RPC通讯，直接批处理可能略微高效（但差异不大）。
- *LS-DYNA* 支持性：LS-DYNA 无Python API，只能通过批处理运行，因此方案B对LS-DYNA是唯一选择。

方案B缺点：

- 结果解析负担：批处理结束后，系统需要解析文本日志或输出文件来判断成败、提取结果。这需要针对不同软件解析不同格式，工作量大且不如API直接获得结构化数据方便。例如，要知道Fluent是否收敛，需要读它的输出日志查特定行；要提取MAPDL结果需解析 `output.out` 的表格。解析过程容易受格式变动或多语言影响（例如软件可能输出英文“DONE”或本地化语言）。
- 缺乏交互控制：一旦批处理开始，除非自行实现监控，否则难以及时干预。比如中途若发现不收敛也很难自动调整参数（除非写自监控脚本）。而API模式下可以每迭代检查。但是对于大多数任务，这点影响不大。
- 脚本兼容性问题：Fluent的Journal若用GUI记录法，版本升级可能失效，因此需要TUI Scheme指令编写[18]；MAPDL的APDL宏也需要考虑不同版本输出略有差异。总之维护这些命令脚本需要一定经验。
- 后处理图形：要在无图形界面下自动保存高质量图片并非易事。例如Fluent/ANSYSAPDL在无GUI模式下渲染复杂场图可能有限制。这需要测试调整，或者如前述采用DPF/PyVista替代。

方案B适用性总结：

- *Fluent*：完全支持批处理。对于不需要复杂交互控制的仿真，编写标准Journal足以自动完成计算并导出需要的数据[16]。尤其在已有大量既定流程和现成Journal模版的情况下，批处理非常高效。唯一挑战是复杂后处理如动画、流线图等，这时候可能结合Ensight或PyFluent更好。但输出收敛曲线、截面数据等用TUI命令即可实现。
- *LS-DYNA*：批处理是既定方式。LS-DYNA输入是关键字文件，本身就定义了所有过程。因此任务执行就是一次性跑到底。结果由输出文件提供，必须通过脚本解析或DPF提取。建议对LS-DYNA一定使用**批处理+DPF**组合：批处理负责运行，DPF负责解析结果，因为没有比DPF更直接读取d3plot的办法了[13]。
- *MAPDL*：批处理模式成熟可靠。如果仿真流程固定（比如只做一次solve即可），APDL脚本能自然表达；且MAPDL输出的“***ANSYS RUN COMPLETED***”等标志易于抓取[21]。后处理用APDL可以输出文本数据，但若要生成图片用批处理直接绘图稍显麻烦，此处可结合DPF/PyMAPDL辅助。总体而言，MAPDL简单场景批处理足矣，复杂场景则API更灵活。

成功与失败的判定机制

无论采用何种执行方式，**可靠地判断仿真任务是否成功完成**至关重要。系统需要自动识别仿真是否正常结束、是否产生有效结果，以及在失败情况下提取错误信息反馈给用户。我们将在此讨论各仿真软件的判定机制实现：

- **进程返回码**：大多数软件进程在正常完成时会返回0码，异常终止则返回非0。FastAPI后台运行子进程（或API调用）时应获取退出码作为初步判断依据。例如，使用 Python 的 `subprocess.run()` 可返回 `CompletedProcess.returncode`。若returncode不为0，则基本确定失败。不过需要注意，有些情况下求解可能遇到收敛错误但进程仍返回0（认为运行"完成"只是结果不收敛）。因此还需进一步分析日志内容。

- **日志/输出文件扫描**：解析仿真软件的输出日志文件或控制台输出，查找特定关键词。

- *Fluent*：Fluent在日志中可能出现"Error"字样（如浮点异常）或"Divergence detected"等信息表明失败。正常完成通常会返回到命令提示符或者打印类似"Fluent completed"之类的消息（Fluent本身无明显"run completed"行，但journal结束即为完成）。可以在Journal最后加一句自定义打印，如Scheme里 `(printf "FLUENT_RUN_COMPLETED")`，然后日志中搜索这行确认执行到末尾。
- *LS-DYNA*：LS-DYNA 在 `.mesg` 或 `.out` 文本里**最后通常有** "正常终止" 提示。例如会有 "**NORMAL TERMINATION**" 或 "**NORMAL TERMINATION, EXIT 0**" 等字样表示成功结束 [22]。如果是错误终止，则可能打印 "ERROR TERMINATION" 或中间出现 "Is aborted" 等错误信息。一个可靠方法是在LS-DYNA结束后**搜索输出日志**：如果找到 "NORMAL TERMINATION" 则标记成功，否则如果出现 "error termination" 则标记失败 [22]。上文LS-OPT文档示例甚至给出了shell脚本通过 grep 'NORMAL TERMINATION' 判断正常结束的做法 [22]。因此，我们在Python中可读取 `.out` 文件的最后若干行来检查。
- *MAPDL*：ANSYS MAPDL 在批处理输出（`-o output.out`）中，若正常结束会有一行类似 "***ANSYS ENGINE FINISHED***" 或 "***ANSYS RUN COMPLETED***" [21]。如果异常终止，通常会出现 "ERROR"、"***ERROR***" 或SIG异常堆栈。判断方法是扫描 output.out：找到 "RUN COMPLETED" 判定成功，没有则失败。此外 MAPDL 在失败时进程返回码往往非0，可配合判断。PyMAPDL 调用则会在异常时抛出Python异常可捕获。

- **结果文件生成**：另一个佐证是检查**预期结果文件**是否生成且大小正常。例如Fluent应输出.cas/.dat或其H5，LS-DYNA应产生d3plot等。如果连结果文件都没有，肯定是失败。但结果文件存在也不一定成功（可能非收敛退出仍写了一部分结果）。因此结合日志关键字更保险。

- **自定义标志**：如前述，对于Fluent，可以在journal最后输出特定字符串，然后监视日志中出现该字符串则说明执行了结尾步骤。对于MAPDL，也可在APDL脚本最后加 `*MSG, FINISHED` 输出自定义信息。在API方案中，也可以在任务结束时由Python主动记录状态。

结合以上，多重判断确保准确。例如任务结束时，系统执行以下逻辑：

```
if process.returncode != 0:
    status = "failed"
elif not log_contains_success_flag(logfile):
    status = "failed"
```

```
else:
    status = "completed"
```

并在失败情况下进一步调用`extract_error_message(logfile)`函数，从日志中抓取错误原因几行文字（例如找到“ERROR”所在行及相邻行），存储以供前端展示给用户。

后处理与结果导出

自动后处理模块负责在仿真完成后，按照预先设定的规则生成**结构化数据**（如CSV、JSON）和**可视化图片**（PNG）等结果文件。根据需求，后处理应支持：多个结果集、灵活的格式、以及大数据量的高效处理。以下逐项说明：

结果数据提取与格式

CSV导出： CSV 是最常见的纯文本表格格式，方便用户在Excel或编程中查看。对于仿真结果，CSV通常用于导出**时History数据**（例如某传感点的温度随时间，某监测面的力随迭代）、或**场分布数据**（如某截面上各点压力值列表）。每种仿真类型都应确定哪些数据适合CSV：

- *Fluent*：可以输出**收敛监控数据**（残差随迭代，用列表示迭代步和不同残差值），**监控点/线数据**（Fluent允许设置Monitor Points，记录随时间的物理量[23]），或者**剖面场数据**（例如在Journal里用`/file/export ascii`命令将某截面上的速度场以CSV导出）。Fluent TUI命令`report/forces`也可输出力值等，可以定向重定向到文件。PyFluent 则可以直接通过API取数据后用pandas存CSV。

- *LS-DYNA*：LS-DYNA若在输入中指定了ASCII输出，如`*DATABASE_NODOUT`(节点输出)，`*DATABASE_GLSTAT`(全局统计)等，会生成CSV格式的文件（以空格/逗号分隔，含列标题）。例如`nodout`文件每行给出每个时间步各节点变量，可直接后缀加.csv方便Excel打开。对于没有预先请求的特定数据，我们可以利用DPF读取d3plot，在Python中获取需要的数据然后写CSV。例如提取某节点集合的位移历程：DPF提取每个时间的位移并存储。**JSON**也是可考虑的格式，适合层次数据（如不同部分的能量统计），但CSV对多数工程师更直观。

- *MAPDL*：APDL可以通过`*EXPORT`将表格数据直接导出为CSV或Excel格式。例如`*EXPORT,tabname,CSV,,filename.csv`可以把定义好的表或数组写成CSV文件。若用PyMAPDL或DPF，则干脆在Python里拿到 NumPy array 后用`numpy.savetxt`写CSV。典型的，如导出所有节点的应力、坐标等表格。在无DPF情况下，可以让APDL在`output.out`里打印结果然后解析，但那不如直接CSV方便。Excel格式（XLSX）也可考虑，用Python的openpyxl等库写入，或APDL的`*EXPORT, , Xls`导出，但XLSX不是纯文本，不便版本管理，因此CSV足矣。

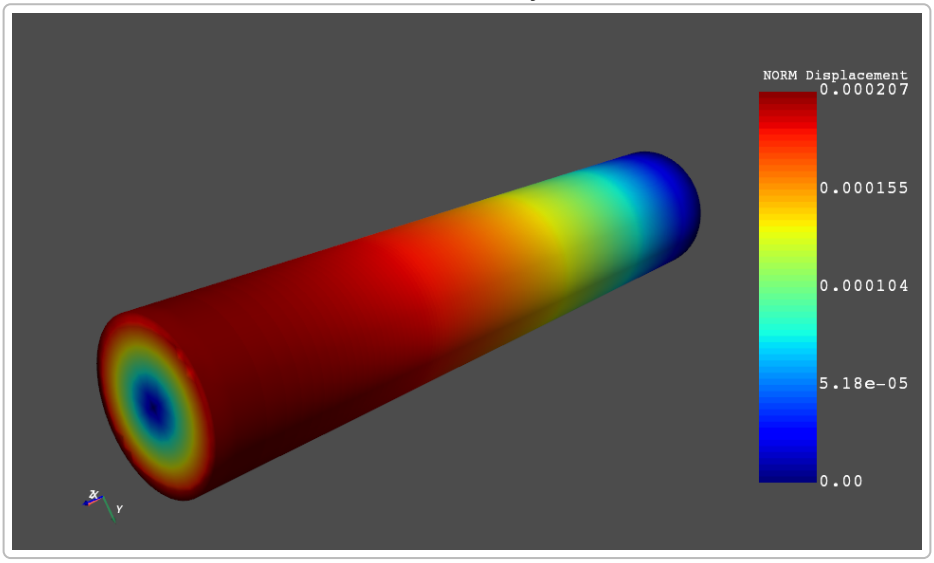
PNG图像： PNG图像用于给用户直观展示结果场。每种仿真类型可生成的典型图片有：

- *Fluent*：流场的**等值面/切面云图**（如压力云图、速度流线图）、**残差收敛曲线**（残差随迭代的折线图）、**监控量曲线**（如流量随时间曲线）。Fluent本身不含高级绘图库，但可以通过**ANSYS EnSight**或**FieldView**等输出。如使用PyFluent/Visualization，则能借助PyVista把求解网格和场变量加载并渲染为PNG[7]。也可在Fluent GUI录制宏，实现比如显示某截面的填色等然后`/display/save`出图。

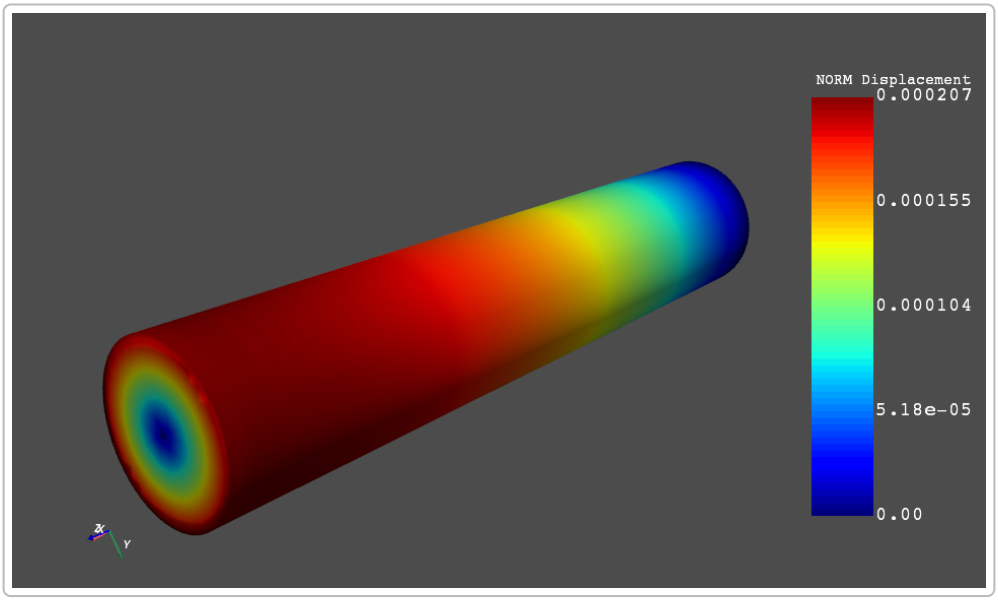
- *LS-DYNA*：结构/冲击仿真的**变形云图**、**应力云图**等。LS-DYNA结果通过 LS-PrePost 或 DPF+PyVista 实现可视化。推荐DPF路线：DPF读取 LS-DYNA 的d3plot，然后封装到 PyVista网格对象，映射应力值，最后渲染

图片。PyDPF-Post 甚至直接提供一行 `.plot()` 绘制等值云的功能 [24] [25]。例如 DPF的示例中，对一个simulation调用 `simulation.stress_eqv_von_mises_nodal()` 得到场结果，再 `.plot()` 即可生成云图 [25]。**注意：**LS-DYNA瞬态过程可能需要生成**动画**（多个时间帧）。此方案中，可以输出为一系列PNG帧，让用户自行组合，或输出一段MP4动画，但自动化生成MP4较复杂，可后续扩展。

- *MAPDL*：结构场的**位移变形图**、**应力分布图**等。PyMAPDL 允许直接 `plot_nodal_displacement()` 等，会调用 vtk 绘图并保存截图 [26]。例如下图展示了通过PyMAPDL脚本生成的一个位移等值云图示例



。该图形显示了一根圆柱结构在机械载荷下的变形和位移分布，由Python脚本从MAPDL结果中提取数据并自动渲染得到。通过类似方式，系统能够在仿真完成后无人工介入地输出PNG格式的结果云图供用户查看。这种方法利用了PyMAPDL/DPF与PyVista的集成，可以获得与ANSYS自带后处理相当的可视化效果 [7]。



上图：通过*PyMAPDL/DPF*接口自动生成的位移云图示例。利用仿真结果文件，*Python*脚本提取节点位移并渲染得到*PNG*图片，展示了结构变形的分布情况。

如果PNG生成存在困难（比如没有安装GUI环境），可退而求其次输出 **VTK** 文件或 **Xdmf** 文件。这些文件可在ParaView等工具中打开，用户可自行查看3D场结果。对于CFD，还可以输出 Ensight格式（.case） [19] 供后续专业后处理。**Excel** 可作为多表数据的容器，例如将多个CSV打包进一个XLSX供下载。

多结果集的处理

“多结果集”是指一次仿真可能产生多个阶段的结果，例如瞬态仿真的各时间步，或参数化仿真的不同参数工况结果。系统需要能够**识别**这些多结果，并允许用户选择感兴趣的部分导出。

具体实现：

- **识别阶段/步**：利用结果文件或日志信息确定有多少独立结果集。例如：
- Fluent瞬态：Fluent会在每个指定步保存一个数据文件或写入 `*.dat.h5` 中，可通过**时间步索引**来区分。DPF 读取 Fluent `.cas.h5` 时，应该可以获取全部 time sets [13] 。如果通过自有手段，可让Fluent在各关键时刻输出中间结果文件（如 `case1.dat, case2.dat` 等）。系统可扫描输出目录特定命名文件来确定时间步列表。
- LS-DYNA瞬态：d3plot文件包含所有时间步，但也可以设置输出间隔。DPF可以读取 LS-DYNA 的多个状态步，并提供按状态索引提取场量的功能。也可在LS-DYNA输入中用 `*DATABASE_BINARY_D3PLOT` 控制每n步输出一个plot文件。例如多个plot文件或单个plot包含多个状态。使用 DPF 则可简单地枚举 simulation.results_sets 来获取有多少 step [24] 。

- MAPDL多工况：例如频率响应分析有多个频率点结果，这种场景下结果文件 `.rst` 已经包含多套结果（Load Step/ Substep）。DPF读取Mechanical结果时每套结果有唯一的 set_id [24] 。可以使用 DPF 或 PyMAPDL 的 post_processing 列出所有结果集，比如 `mapdl.post_processing.list_available_results()`.

- **在Web界面呈现**：后台收集这些结果集信息（如时间=0.1s,0.2s...或LoadStep1-5等），通过API传给前端。前端可让用户勾选需要导出的结果集。例如提供一个多选框列表列出“Time Step 1 (0.0s), Time Step 2 (0.01s), ...”或“Mode 1, Mode 2, ...”等。

- **按需导出**：当用户选择了所需结果集后，再触发后处理导出。例如用户可能只关心最终一步，则无需导出所有步数据/图片，减少开销。实现上，可以在仿真完成后暂存**结果对象或文件**，待用户请求时再进行提取：

- 如果结果文件易于随机访问（如DPF可以按需读取特定步的数据），则每次用户请求时读取对应步并生成CSV/PNG。
- 如果不易随机访问，则可以在仿真完成时先生成所有步的数据文件，让用户按需下载。鉴于存储和带宽考虑，倾向前者。

例如，借助 PyDPF，我们可以做到：仿真完成后不马上生成任何CSV，只是在用户点击“导出选中结果”时，后台脚本执行：加载结果 -> 提取对应时间步或载荷步的所需量 -> 生成CSV/PNG -> 返回或提供下载链接。

这样的按需后处理也可以防止大量不必要的数据传输，提高效率。

全流程框架设计

综上分析，我们确定采用**模块化**设计，将各环节职责清晰划分。下面给出全流程框架的设计要点以及部分模块说明：

1. **FastAPI 接口层**：包括路由和请求处理函数。
2. `POST /upload_task`：处理任务上传，请求体包含文件和参数JSON。逻辑：保存文件 -> 根据扩展识别类型 -> 初始化任务对象 -> 检查参数合法(许可) -> 将任务加入执行队列，返回任务ID。
3. `GET /task_status/{task_id}`：查询任务状态，返回状态枚举（queued/running/completed/failed）及进度信息（如已运行时间、剩余预估等，如果可算出）。
4. `GET /task_result_list/{task_id}`：在任务完成后，获取该任务可用结果集列表及文件列表。如多结果集则列出可选项。
5. `POST /export_results/{task_id}`：用户提交要导出的结果集选项，服务器生成对应文件压缩包或下载链接。

6. 其他如通过WebSocket推送实时日志，可以拓展。

7. **任务管理和调度**：考虑到Python的多线程对CPU密集任务效果不佳，可使用**多进程或异步IO**。简化起见，这里示范用Python内置 `concurrent.futures.ProcessPoolExecutor` 或者FastAPI自带 `BackgroundTasks` 来异步运行任务。实际生产中，**Celery**+消息队列或**自定义调度服务**更健壮。

8. 任务管理器维护一个 `dict` 映射 task_id -> Task 对象/状态。Task对象包含属性：id、type、status、progress、result_path等。
9. 提交任务时生成唯一ID（可用UUID），创建Task对象放入字典，然后在线程/进程池中启动其执行函数。
10. 执行过程中Task.status更新为running，完成则 success/failed；progress可在仿真日志解析（比如时间步迭代）中更新。

11. 任务完成后，结果文件存储在预定义位置，Task对象中记录结果列表。

12. **仿真任务基类和子类**：使用**面向对象**封装不同仿真类型逻辑。定义抽象基类 `SimulationTask`，主要方法：

13. `prepare()`：准备运行环境，比如设置工作目录、编写输入文件/脚本（如Fluent journal, MAPDL macro）。
14. `run()`：执行仿真主过程（通过API或subprocess），实时收集日志输出用于监控。
15. `post_process()`：在仿真完成后调用，进行统一的结果处理（可以调用DPF或解析输出文件），生成结果文件列表。
16. `check_success()`：检查仿真是否成功（基于返回码和日志）。
17. 以及辅助如 `update_progress()` 用于在日志解析中更新进度。

对于每种仿真类型（FluentTask, LsdynaTask, MapdlTask），实现各自的上述方法逻辑。

1. **日志及错误处理**：将仿真输出实时写入日志文件，并在界面提供查看。出错时捕获异常写入Task对象的 error_message 字段。

2. **结果存储**：可按task分类存储结果文件，如 `{work_dir}/{task_id}/` 下保存上传的原始文件、副本、日志、结果CSV/PNG。这样便于下载打包和清理。

接下来，通过代码示例片段展示上述模块的实现思路：

(1) SimulationTask 抽象基类定义

```
import subprocess, os, shutil
from abc import ABC, abstractmethod

class SimulationTask(ABC):
    def __init__(self, task_id: str, files: list, params: dict, config: dict):
        self.id = task_id
        self.files = files        # list of file paths
        self.params = params      # user-specified or default parameters
        self.config = config      # loaded config for this sim type
        self.work_dir = os.path.join(config['global']['work_dir'], task_id)
        self.status = "queued"
        self.progress = 0.0
        self.result_files = []    # will hold paths of result outputs
        self.error_message = ""
        # Ensure work directory exists
        os.makedirs(self.work_dir, exist_ok=True)
        # Move uploaded files into work_dir for easier access
        for f in self.files:
            shutil.move(f, os.path.join(self.work_dir, os.path.basename(f)))
        self.files = [os.path.join(self.work_dir, os.path.basename(f)) for f in
self.files]

    def update_progress(self, value: float):
        # Update progress (0.0 to 1.0) and possibly log it
        self.progress = value

    @abstractmethod
    def prepare(self):
        """Prepare necessary input files or scripts for simulation."""
        pass

    @abstractmethod
    def run(self):
        """Run the simulation (either via API or subprocess)."""
        pass

    @abstractmethod
    def post_process(self):
        """Process results to produce CSV/PNG outputs."""
        pass
```

```
    @abstractmethod
    def check_success(self):
        """Determine if simulation succeeded, set error_message if any."""
        pass
```

(2) FluentTask 子类实现示例

```
class FluentTask(SimulationTask):
    def prepare(self):
        # Assume self.files includes a .cas (and possibly .dat)
        cas_file = None
        dat_file = None
        for f in self.files:
            if f.endswith('.cas') or f.endswith('.cas.h5'):
                cas_file = os.path.basename(f)
            elif f.endswith('.dat'):
                dat_file = os.path.basename(f)
        # Compose a Fluent journal file
        journal_path = os.path.join(self.work_dir, "run.jou")
        with open(journal_path, 'w') as jou:
            jou.write(f'/file/set-workspace "{self.work_dir}"\n')
            # Read case (and data if exists)
            if cas_file and dat_file:
                jou.write(f'/file/read-case-data "{cas_file} {dat_file}"\n')
            elif cas_file:
                jou.write(f'/file/read-case "{cas_file}"\n')
            # Apply default or user settings (e.g., iterations)
            iters = self.params.get('iterations', 100)
            # If transient vs steady could be decided by .cas content or user
param
            # Here assume steady for simplicity:
            jou.write(f'/solve/initialize/initialize-flow\n')
            jou.write(f'/solve/iterate {iters}\n')
            # After solve, export results - for example pressure field on
boundary or residuals
            jou.write(f'/plot/print-to-file residuals.csv\n')  # hypothetical
command to output residuals
            jou.write(f'/file/export/ascii data.csv all pressure velocity ()
"\n')
            # The above line is illustrative; actual Fluent export syntax
differs
            jou.write('\nexit\nyes\n')
        self.journal_file = journal_path

    def run(self):
```

```
        self.status = "running"
        # Build Fluent command
        solver = self.config['solver_path']
        dim_flag = self.config['default'].get('dim', '3ddp')  # use default or
user-specified precision
        cores = self.params.get('cores', self.config['default']['cores'])
        cmd = f'"{solver}" {dim_flag} -g -t{cores} -i "{self.journal_file}"'
        # Redirect output to a log file
        log_path = os.path.join(self.work_dir, "fluent_output.log")
        with open(log_path, 'w') as logf:
            proc = subprocess.Popen(cmd, shell=True, cwd=self.work_dir,
                                    stdout=logf, stderr=subprocess.STDOUT)
            # Optionally, one could read stdout line by line here to update
progress.
            proc.wait()
            self.returncode = proc.returncode
        # After run complete, check success and proceed
        self.log_file = log_path

    def check_success(self):
        # Basic check: return code
        if hasattr(self, 'returncode') and self.returncode != 0:
            self.error_message = f"Fluent exited with code {self.returncode}"
            return False
        # Further check log for errors or completion
        success = False
        with open(self.log_file, 'r') as f:
            lines = f.readlines()
            for ln in lines[-10:]:  # check last 10 lines
                if "Error" in ln or "error" in ln:
                    self.error_message += ln
            # Look for a sign that solve iteration completed
            for ln in lines:
                if "exiting Fluent" in ln or "FLUENT_RUN_COMPLETED" in ln:
                    success = True
                    break
        if not success and not self.error_message:
            self.error_message = "Fluent run did not complete normally."
        return success

    def post_process(self):
        # For Fluent, we might have already exported CSV in journal.
        # Check if residuals.csv or data.csv created, if so add to result_files.
        exported_files = ["residuals.csv", "data.csv"]
        for fname in exported_files:
            fpath = os.path.join(self.work_dir, fname)
            if os.path.exists(fpath):
                self.result_files.append(fpath)
```

```
        # Additionally, we could generate images via PyFluent or DPF if needed
        # For example, use PyDPF to read cas/dat and plot a contour.
        # Skipping actual DPF code for brevity; assume an image was created:
        # dpf_image_path = os.path.join(self.work_dir, "contour.png")
        # self.result_files.append(dpf_image_path)
```

说明： 上述 `FluentTask` 为示例简化版，实际应用中 Fluent TUI 导出命令应使用正确的语法（例如 `/file/export ascii` 的具体用法需要选定区域和变量）。这里演示如何拼接命令。`check_success()` 函数通过检查进程返回码和日志中关键字来判断结果。[22] 此外，可根据需要扩展如读取 Fluent 日志中最后一行是否包含"正常结束"提示（Fluent没有明确提示，我们可以依赖journal结束标志）。`post_process()` 假设Journal已经导出了一些结果文件并将其加入列表。如果需要更多处理，例如绘图，则可以调用PyFluent或DPF，此处留白。

(3) LS-DYNA Task 子类实现简要

```
class LsdynaTask(SimulationTask):
    def prepare(self):
        # LS-DYNA input is presumably one .k file (or multiple included files).
        # Ensure the main .k file is identified:
        k_files = [f for f in self.files if f.endswith('.k') or
f.endswith('.key')]
        if not k_files:
            raise RuntimeError("No LS-DYNA .k input found")
        self.main_k = os.path.basename(k_files[0])
        # LS-DYNA requires no special preparation beyond having the file,
        # but we could set environment or license if needed.

    def run(self):
        self.status = "running"
        solver = self.config['solver_path']
        cores = self.params.get('cores', self.config['default']['cores'])
        mem = self.params.get('memory', self.config['default']['memory'])
        cmd = f'"{solver}" i={self.main_k} memory={mem}m ncpu={cores}'
        log_path = os.path.join(self.work_dir, "dyna.out")
        with open(log_path, 'w') as lf:
            proc = subprocess.Popen(cmd, shell=True, cwd=self.work_dir,
                                    stdout=lf, stderr=subprocess.STDOUT)
            proc.wait()
            self.returncode = proc.returncode
        self.log_file = log_path

    def check_success(self):
        if hasattr(self, 'returncode') and self.returncode != 0:
            self.error_message = f"LS-DYNA exited with code {self.returncode}"
            return False
        success = False
        with open(self.log_file, 'r') as f:
```

```
            text = f.read()
            # LS-DYNA might have all output in .out or .mesg. Check both if
exist.
        # Check for normal termination message
        if "NORMAL TERMINATION" in text:
            success = True
        if "error termination" in text or "ERROR TERMINATION" in text:
            success = False
            # capture snippet around error
            idx = text.lower().find("error termination")
            self.error_message = text[idx: idx+100] if idx!=-1 else "LS-DYNA
error termination."
        if not success and not self.error_message:
            self.error_message = "LS-DYNA did not terminate normally."
        return success

    def post_process(self):
        # Use PyDPF to read d3plot and extract results.
        d3plot_path = None
        # Find d3plot (there could be multiple files like d3plot, d3plot01 etc
if segmented)
        for fname in os.listdir(self.work_dir):
            if fname.startswith("d3plot"):
                d3plot_path = os.path.join(self.work_dir, fname)
                break
        if d3plot_path:
            from ansys.dpf import core as dpf_core
            model = dpf_core.Model(d3plot_path)
            # Example: get von Mises stress at last time step on all nodes
            # (DPF by default might take last set if not specified)
            stress_op = dpf_core.operators.results.stress_eqv(model)
            stress_field = stress_op.outputs.fields_container()[0]
            # Save to CSV: node id and value
            data = [(eid, val) for eid, val in zip(stress_field.scoping.ids,
stress_field.data)]
            csv_path = os.path.join(self.work_dir, "vonMises_last.csv")
            with open(csv_path, 'w') as cf:
                cf.write("EntityID, VonMisesStress\n")
                for eid, val in data:
                    cf.write(f"{eid},{val}\n")
            self.result_files.append(csv_path)
            # Also we can produce an image of the stress contour:
            try:
                import pyvista as pv
                mesh = model.metadata.meshed_region  # get mesh from DPF
                # Create PyVista mesh and add data
                pvmesh = pv.UnstructuredGrid(mesh.nodes.coordinates,
mesh.connectivity, mesh.cell_types)
```

```
                pvmesh.point_data["vonMises"] = stress_field.data
                plotter = pv.Plotter(off_screen=True)
                plotter.add_mesh(pvmesh, scalars="vonMises", cmap="jet")
                plotter.view_xy()  # adjust view as needed
                img_path = os.path.join(self.work_dir, "vonMises_last.png")
                plotter.screenshot(img_path)
                self.result_files.append(img_path)
            except Exception as e:
                # If any issue (like pyvista not installed), skip image
                pass
        # If other ascii outputs (glstat, etc) exist, just include them
        for fname in os.listdir(self.work_dir):
            if fname.lower().endswith('.csv') or
fname.lower().endswith('.glstat'):
                self.result_files.append(os.path.join(self.work_dir, fname))
```

说明：`LsdynaTask` 在 `post_process()` 中演示了如何使用ANSYS DPF来读取LS-DYNA生成的 `d3plot` 文件并提取结果。根据 PyDPF 文档，DPF 已支持 LS-DYNA 的结果文件 [13]。代码中构造 DPF Model，使用预定义算子计算了等效应力场，然后将最后一步的结果输出为CSV，并尝试用 PyVista 绘制云图保存为PNG。这与上文分析的**方案组合**一致：求解用批处理，后处理利用DPF/Python获取结构化结果。

(4) MAPDLTask 子类实现简要

```
class MapdlTask(SimulationTask):
    def prepare(self):
        # Identify main input (.inp or .cdb)
        inp_file = None
        for f in self.files:
            if f.endswith('.inp') or f.endswith('.mac'):
                inp_file = os.path.basename(f)
        if not inp_file:

# If only a .cdb given, we might need to create an input to read cdb and solve
            cdb_file = next((f for f in self.files if f.endswith('.cdb')), None)
            if cdb_file:
                inp_file = "auto_input.inp"
                with open(os.path.join(self.work_dir, inp_file), 'w') as fi:
                    fi.write("/batch\n")
                    fi.write(f"/input,{os.path.basename(cdb_file)}\n")
                    fi.write("/solu\nsolve\nfinish\n")
# very basic solve, real script would contain loads etc.
                    fi.write("/
post1\nset,last\nprnsol,u,x\nprnsol,u,y\nprnsol,u,z\nfinish\n")
            else:
                raise RuntimeError("No MAPDL input provided")
        self.input_file = inp_file
```

```
    def run(self):
        self.status = "running"
        solver = self.config['solver_path']
        cores = self.params.get('cores', self.config['default']['cores'])
        lic = self.params.get('product', self.config['default'].get('product',
'ansys'))
        cmd = f'"{solver}" -b -p {lic} -np {cores} -i "{self.input_file}" -o
"output.out"'
        proc = subprocess.Popen(cmd, shell=True, cwd=self.work_dir)
        proc.wait()
        self.returncode = proc.returncode
        self.log_file = os.path.join(self.work_dir, "output.out")

    def check_success(self):
        if hasattr(self, 'returncode') and self.returncode != 0:
            self.error_message = f"MAPDL exited with code {self.returncode}"
            return False
        success = False
        with open(self.log_file, 'r') as f:
            text = f.read()
            if "RUN COMPLETED" in text or "NORMAL COMPLETION" in text:
                success = True
            if "*** ERROR ***" in text:
                # capture the first error line
                idx = text.index("*** ERROR ***")
                self.error_message = text[idx: idx+100].splitlines()[0]
        if not success and not self.error_message:
            self.error_message = "MAPDL did not complete successfully."
        return success

    def post_process(self):
        # Approach: use PyDPF to read .rst
        rst_file = None
        for fname in os.listdir(self.work_dir):
            if fname.endswith('.rst'):
                rst_file = os.path.join(self.work_dir, fname)
                break
        if rst_file:
            from ansys.dpf import core as dpf
            model = dpf.Model(rst_file)
            # e.g., extract displacement field for all nodes in last result set
            disp_op = dpf.operators.result.displacement(model)
            disp_fc = disp_op.outputs.fields_container()
            # fields_container could have multiple sets, we take the last one:
            last_field = disp_fc[-1]
            # Save nodal displacement to CSV
            csv_path = os.path.join(self.work_dir, "displacement.csv")
```

```
            with open(csv_path, 'w') as cf:
                cf.write("NodeID, Ux, Uy, Uz\n")
                ids = last_field.scoping.ids
                data = last_field.data.reshape((len(ids), -1))
                for nid, vals in zip(ids, data):
                    cf.write(f"{nid}, {vals[0]:.6e}, {vals[1]:.6e}, {vals[2]:.
6e}\n")
            self.result_files.append(csv_path)
            # Also generate a plot (using DPF's plot or PyVista)
            try:
                import pyvista as pv
                mesh = model.metadata.meshed_region
                pvmesh = pv.UnstructuredGrid(mesh.nodes.coordinates,
mesh.connectivity, mesh.cell_types)
                # assume nodal displacement magnitude
                import numpy as np
                U = np.linalg.norm(data, axis=1)
                pvmesh.point_data["U_mag"] = U
                plotter = pv.Plotter(off_screen=True)
                plotter.add_mesh(pvmesh, scalars="U_mag", cmap="jet")
                plotter.view_xy()
                img_path = os.path.join(self.work_dir, "displacement.png")
                plotter.screenshot(img_path)
                self.result_files.append(img_path)
            except Exception:
                pass
        # If .out contains PRNSOL tables as fallback
        with open(self.log_file, 'r') as f:
            lines = f.readlines()
        # parse PRNSOL output if needed (omitted for brevity)
```

说明：`MapdlTask` 展示了调用ANSYS Mechanical APDL 批处理并利用 DPF 读取 `.rst` 结果文件的流程[12]。`post_process()` 提取最后一步所有节点位移并保存 CSV，同时尝试绘制位移幅值云图为 PNG。这里采用了与LS-DYNA相同的PyVista流程。对于多结果集(如多工况)，DPF的 fields_container 会包含多个 field，可以让用户选择不同 set 导出（此处简单取最后一个作为示例）。`check_success()` 判断 output.out 是否包含 “RUN COMPLETED” 等字样[21]，错误则查找 `*** ERROR ***` 提示。

(5) FastAPI 接口整合

最后，将上述Task整合进FastAPI接口。下面是FastAPI部分伪代码：

```
from fastapi import FastAPI, UploadFile, File, BackgroundTasks
from uuid import uuid4

app = FastAPI()
```

```
# In-memory storage for tasks (task_id -> task object)
tasks = {}

def run_task(task):
    """Function to run in background for executing simulation task."""
    try:
        task.prepare()
        task.run()
        success = task.check_success()
        if success:
            task.post_process()
            task.status = "completed"
        else:
            task.status = "failed"
        # If needed, we can log or notify here
    except Exception as e:
        task.status = "failed"
        task.error_message = str(e)

@app.post("/upload_task")
async def upload_task(files: list[UploadFile] = File(...), sim_type: str = "",
params: dict = None, background_tasks: BackgroundTasks = None):
    # Save uploaded files to a temp location
    saved_paths = []
    for file in files:
        content = await file.read()
        path = os.path.join("/tmp", file.filename)
        with open(path, "wb") as f:
            f.write(content)
        saved_paths.append(path)
    # Determine sim_type if not provided
    if not sim_type:
        # simple logic: check extensions
        for p in saved_paths:
            ext = os.path.splitext(p)[1].lower()
            if ext in [".cas", ".dat", ".cas.h5"]:
                sim_type = "fluent"
                break
            elif ext in [".k", ".key"]:
                sim_type = "lsdyna"
                break
            elif ext in [".inp", ".cdb", ".mac"]:
                sim_type = "mapdl"
                break
    if sim_type not in ["fluent", "lsdyna", "mapdl"]:
        return {"error": "Unrecognized simulation type"}
    # Load corresponding config
    sim_config = config['simulation'][sim_type]
```

```
    # Merge default params
    full_params = {**sim_config.get('default', {}), **(params or {})}
    # License check
    max_cores = sim_config.get('license', {}).get('max_cores')
    if max_cores and full_params.get('cores', 1) > max_cores:
        return {"error": f"Requested cores exceed license limit ({max_cores})"}
    # Create task instance
    task_id = str(uuid4())
    if sim_type == "fluent":
        task = FluentTask(task_id, saved_paths, full_params, {**sim_config,
**config.get('global', {})})
    elif sim_type == "lsdyna":
        task = LsdynaTask(task_id, saved_paths, full_params, {**sim_config,
**config.get('global', {})})
    elif sim_type == "mapdl":
        task = MapdlTask(task_id, saved_paths, full_params, {**sim_config,
**config.get('global', {})})
    tasks[task_id] = task
    # Launch background execution
    background_tasks.add_task(run_task, task)
    return {"task_id": task_id, "status": task.status}

@app.get("/task_status/{task_id}")
def get_status(task_id: str):
    task = tasks.get(task_id)
    if not task:
        return {"error": "Task not found"}
    return {
        "status": task.status,
        "progress": task.progress,
        "error_message": task.error_message if task.status == "failed" else None
    }

@app.get("/task_result_list/{task_id}")
def get_result_list(task_id: str):
    task = tasks.get(task_id)
    if not task:
        return {"error": "Task not found"}
    if task.status != "completed":
        return {"error": "Task not completed yet"}
    # If multiple result sets exist, list them (this example assumes results
already generated in files)
    results = []
    for fpath in task.result_files:
        fname = os.path.basename(fpath)
        results.append({"file": fname})
    return {"results": results}
```

```python
@app.get("/download_result/{task_id}/{filename}")
def download_file(task_id: str, filename: str):
    task = tasks.get(task_id)
    if not task:
        return {"error": "Task not found"}
    file_path = os.path.join(task.work_dir, filename)
    if not os.path.isfile(file_path):
        return {"error": "File not found"}
    # Use Starlette FileResponse for streaming download
    return FileResponse(path=file_path, filename=filename)
```

上述FastAPI接口实现了上传、查询状态、列结果和下载结果的基础功能。用户通过 `/upload_task` 提交任务后，后端立即返回 task_id 并开始后台执行。用户可反复调用 `/task_status/{task_id}` 查看进展。当状态变为 `completed` 时，可以通过 `/task_result_list/{task_id}` 获取生成的结果文件列表，然后调用 `/download_result/{task_id}/{filename}` 下载具体文件。前端也可提供界面直接展示图片或表格内容（比如CSV以表格形式显示）。

需要注意，**多结果集**的处理在此接口中可以进一步扩展：比如 `task_result_list` 不仅返回文件列表，还可返回不同result set的信息（如时间步列表）。也可以设计成 `/export_results` 接口接受参数（如导出哪几个时间步、哪种变量），然后后台动态调用 `task.post_process(selected_sets=...)` 生成相应文件。由于上述 Task.post_process 实现中已经一次性输出了某些结果，为了更贴合"按需导出"，我们可以让 `post_process()` 只收集原始结果对象不输出全部文件，待用户请求时再输出。限于篇幅，这里提供最直接的实现思路。

方案比较与总结

综上，本方案通过结合**Python API**与**批处理**两种技术路线，针对 Fluent、LS-DYNA、MAPDL 三种仿真软件提供了完整的解决方案设计。在实际生产落地中，可以按照以下建议进行取舍：

- **Fluent**：优先考虑使用 PyFluent 实现深度自动化（特别是复杂后处理需求时），但保留批处理Journal方案作为稳妥后备（适合标准求解）。两种方式各有优劣，可根据团队熟悉程度选择。无论哪种方式，Fluent日志与结果导出都相对清晰，成败判定可通过日志或PyFluent异常判断。
- **LS-DYNA**：采用批处理启动求解是既定方式，同时充分利用 DPF 接口进行后处理。这弥补了LS-DYNA缺乏自带后处理导出的不足，实现与其他求解器一致的数据输出格式。成败判定主要依赖日志搜索"NORMAL TERMINATION"等 [22]，并辅以进程返回码。
- **MAPDL**：可灵活选择。对于简单分析，批处理APDL足够解决，结果亦可通过DPF读取，使得即便采用批处理算, 后处理也能用现代方法完成。如果需要与ANSYS Workbench结合或做高级控制，PyMAPDL更为方便。许可检查在MAPDL场景下尤其重要（确保并行核数在许可范围）。

配置管理在本方案中发挥了重要作用，所有可变的路径、命令和参数都集中定义，方便日后维护扩展。另外，**错误处理和日志**机制确保了出现问题时可以快速定位——日志文件既提供给开发者调试，也可以通过前端提供下载或查看功能给予用户，提升可信度。

完整生产级代码应在此基础上考虑更多健壮性因素：如异步任务超时处理（防止仿真挂起无限等待）、磁盘空间管理（删除过期任务文件）、安全性（防止用户上传恶意脚本）等。在初步实现后，可逐步完善。

本方案通过引用官方文档和资料，验证了各项设计决策的可行性，例如：PyFluent的全功能Python控制 [4]、PyMAPDL/DPF对结果的提取 [10] [12]、以及许可限制和正常结束判定的依据 [1] [22]。所提供的模块化代码示例涵盖了核心功能点，在实际部署时可作为基础框架扩展成完整系统。通过该方案，仿真管理系统将能够高效地调度和监控不同类型的仿真任务，并自动产出用户所需的多样化结果格式，极大减少人工干预，提高仿真工作流程的自动化和可靠性。

---

[1] [19] How many cores are supported with a single or multiple ANSYS HPC pack? | Ansys Knowledge
https://innovationspace.ansys.com/knowledge/forums/topic/how-many-cores-are-supported-with-a-single-or-multiple-ansys-hpc-pack/

[2] [3] LS-DYNA — ARC User Documentation 1.0 documentation
https://www.docs.arc.vt.edu/software/lsdyna.html

[4] [5] [6] [7] [14] Providing Open-Source Access to Ansys Fluent with PyFluent
https://www.ansys.com/blog/open-source-access-to-fluent-with-pyfluent

[8] [9] [10] [11] [20] [26] Postprocessing — PyMAPDL
https://mapdl.docs.pyansys.com/version/stable/user_guide/post.html

[12] [13] [24] [25] PyDPF-Post — PyDPF-Post
https://post.docs.pyansys.com/

[15] [16] [17] [18] How to create and execute a FLUENT journal file? | Ansys Knowledge
https://innovationspace.ansys.com/knowledge/forums/topic/how-to-create-and-execute-a-fluent-journal-file/

[21] Big ANSYS model, memory requirements - sci.engr.analysis
https://sci.engr.analysis.narkive.com/wGClqJJA/big-ansys-model-memory-requirements

[22] General — Welcome to LS-OPT Support Site...
https://www.lsoptsupport.com/howtos-4/user-defined-solvers-1/general

[23] CSV | Ansys Knowledge - Ansys Customer Center
https://innovationspace.ansys.com/knowledge/forums/topic-tag/csv/