A Technical Blueprint for an Autonomous Generative Design Agent

Executive Summary

This report presents a comprehensive technical blueprint for transforming the rag-agent-framework into a sophisticated, autonomous engineering design assistant. The proposed architecture establishes a symbiotic relationship between two primary intelligent agents: a "Librarian" agent, responsible for managing a complex, interconnected engineering knowledge base, and a "Design Agent," which intelligently orchestrates a suite of generative tools to create, analyze, and iteratively refine mechanical designs. The core of this system moves beyond conventional Retrieval-Augmented Generation (RAG) by implementing a knowledge graph-powered Librarian, enabling it to synthesize information and answer complex relational queries. The Design Agent operates within a stateful think → propose → simulate → refine loop, functioning as an optimization controller that leverages parametric CAD generation, topology optimization, and finite element analysis (FEA) simulation tools. Furthermore, the integration of specialized machine learning models, such as Mesh Variational Autoencoders (VAEs), provides a mechanism for design exploration, allowing the agent to generate novel concepts. This entire workflow is built upon the CrewAI framework, leveraging its capabilities for agent collaboration and task management. Critically, the architecture incorporates a robust "Digital Thread" for end-to-end traceability, adapting MLOps best practices to ensure every action, decision, and artifact is versioned, logged, and reproducible, meeting the stringent demands of engineering and compliance.

Part I: Fortifying the Knowledge Core - The Engineering Librarian RAG

The foundation of any intelligent engineering assistant is the depth and accessibility of its knowledge. For an agent to answer questions like "What was the wall thickness on pump housing v2?" or "Which materials passed ISO 10993?", it requires more than simple document retrieval. The system must understand the entities (materials, standards, parts) and the relationships between them. This necessitates evolving the RAG framework from a text retriever into a knowledge synthesizer. A simple vector search on raw text is insufficient because engineering knowledge is inherently structured and relational.[1] A query about a material passing a specific standard requires connecting a material entity to a standard entity via a "passed" relationship, information that may be spread across a material data sheet and a separate test report. This insight dictates that the most critical component of the "Librarian" agent's architecture is a sophisticated ingestion pipeline that pre-processes engineering data into a structured knowledge graph, augmenting it with semantic search capabilities.

1.1. Proposed Architecture for the rag-agent-framework

A review of the existing yangvianno/rag-agent-framework suggests a solid starting point. To accommodate the complexity of the engineering domain, a more modular and scalable architecture is recommended. This architecture should be organized into three distinct, decoupled services, a pattern that aligns with best practices for building robust RAG systems.[2]

- **The Ingestion Service:** This service will be a collection of specialized pipelines, each tailored to a specific engineering data type (CAD, PDF, FEA logs). Its responsibility is not merely to load and chunk data but to parse, extract structured information, identify entities (e.g., 'Pump Housing v2', 'ISO 10993', 'AISI 316L Stainless Steel'), and define the relationships between them.
- **The Storage Service:** A hybrid storage model is proposed to capture both structured and unstructured knowledge.
    - **Graph Database:** A graph database like Neo4j will serve as the primary store for the knowledge graph. It will house the nodes (entities) and edges (relationships) extracted by the Ingestion Service. This structure is ideal for representing the complex interconnections within engineering data.
    - **Vector Database:** A vector database such as FAISS (utilized in advanced RAG agents [1]) or a similar high-performance library will store the vector

embeddings of text chunks. Each text chunk will be linked back to its corresponding node in the graph database, allowing the system to retrieve specific text passages associated with an entity.

- **The Retrieval Service:** This will be implemented as a CrewAI tool for the Librarian agent. When the agent receives a natural language query, this tool will intelligently decide whether to translate the query into a formal graph query (e.g., Cypher for Neo4j) to traverse relationships, perform a semantic vector search for relevant context, or a combination of both. This adaptive retrieval strategy mirrors the multi-step reasoning and planning seen in advanced agentic frameworks.[1]

This modular architecture ensures that each component can be developed, scaled, and maintained independently, creating a resilient and extensible foundation for the engineering knowledge base.

1.2. Advanced Ingestion Pipelines for Heterogeneous Engineering Data

The efficacy of the Librarian agent hinges on the quality of the data ingested into its knowledge base. Each data type presents unique parsing challenges that require specialized tools and techniques.

1.2.1. Deconstructing CAD Models (STEP/IGES)

CAD models are rich sources of information, containing not only explicit metadata but also implicit geometric data that is critical for engineering queries. Extracting this information requires libraries capable of deep interrogation of the model's structure.

- **Problem:** A query like "What is the wall thickness?" cannot be answered by parsing file headers. It requires geometric analysis of the 3D model itself.
- **Solution:** A dual-library approach is recommended. For high-level structural parsing and metadata extraction, steputils offers a user-friendly, Pythonic Document Object Model (DOM) interface.[5] For deep geometric analysis, python-occ is the tool of choice, as it provides direct Python bindings to the powerful OpenCASCADE Technology (OCCT) C++ kernel, enabling precise measurements and topological traversal.[7]
- **Implementation Steps:**

1. **Metadata Extraction:** Use steputils.p21.load to parse the STEP file's P21 structure. This allows for the extraction of header information such as FILE_NAME, time_stamp, author, organization, and the governing FILE_SCHEMA.[8] This metadata provides essential provenance for the design artifact.
2. **Geometric Analysis:** Utilize python-occ to load the model and traverse its topological entities (faces, edges, vertices). A function can be written to iterate through the model's shells, identify pairs of opposing faces, and calculate the distance between them to determine wall thickness. Similar methods can be applied to measure hole diameters, fillet radii, and other critical geometric features.
3. **Structuring for RAG:** The extracted information will be structured and stored as properties on a node representing the CAD model in the knowledge graph. For example, a node for pump_housing_v2.step would have attributes like {'part_name': 'pump_housing_v2', 'author': 'J. Doe', 'schema': 'AP242', 'features': [{'name': 'main_outlet', 'wall_thickness': 3.5, 'units': 'mm'}]}. This structured data makes the information directly queryable.

1.2.2. Unlocking Visually Rich Documents (PDFs, Engineering Drawings)

Technical documents, such as specification sheets, test reports, and industry standards, are often delivered as PDFs. These documents are "visually rich," meaning their layout—including tables, diagrams, and columns—is essential to understanding their content.[9] Recent research emphasizes that extracting information from such documents requires layout-aware models that go beyond simple text scraping.[10]

- **Problem:** Standard PDF text extractors often fail to preserve tabular structure and cannot interpret diagrams or engineering drawings, which contain a wealth of information.
- **Solution:** A multi-modal, hybrid approach is necessary.
  1. **Text and Table Extraction:** For general text extraction, pdfplumber is a strong choice due to its detailed access to the position and properties of every character, line, and rectangle on a page, giving fine-grained control.[12] For extracting tables,
     Camelot is the state-of-the-art library.[14] It is specifically designed for this task and offers multiple parsing strategies: the
     Lattice method, which uses explicit line boundaries, is perfect for grid-lined

tables, while the Stream method, which uses whitespace to infer cell layout, can handle less structured tables.[16]
Camelot can export extracted tables directly into pandas DataFrames, a format ideal for further processing.[17]

2. **Advanced Layout and Drawing Understanding:** For pages containing complex engineering drawings, schematics, or diagrams with GD&T (Geometric Dimensioning and Tolerancing) callouts, traditional parsers are inadequate. Inspired by recent academic advancements, this pipeline should incorporate a Vision-Language Model (VLM). Papers have demonstrated success using fine-tuned document understanding transformers like Donut or smaller, efficient VLMs like Florence-2 to parse engineering drawings and output structured JSON.[19] The ingestion pipeline will detect if a PDF page contains a drawing; if so, it will pass the page image to a VLM for interpretation, extracting key information like dimensions, tolerances, and material specifications directly from the drawing.
3. **Structuring for RAG:** Extracted tables will be converted into a structured format (e.g., CSV or JSON), and a link to this structured data file will be stored as a property on the document's node in the knowledge graph. The textual content of the PDF will be chunked, embedded, and stored in the vector database, with each chunk linked back to the document node and page number. The structured JSON output from the VLM will be similarly linked, making the drawing's contents searchable.

1.2.3. Structuring Simulation Data (FEA Logs)

Finite Element Analysis (FEA) logs are a critical record of a design's performance under simulation. While often appearing as unstructured text files, they contain vital, structured information such as convergence status, final stress values, displacement metrics, and error messages.

- **Problem:** The raw, unstructured format of log files makes them difficult to query directly. Key performance indicators must be extracted and structured for analysis.
- **Solution:** For logs with a consistent, repeating format, a custom parser using Python's built-in re module (regular expressions) can be highly effective and efficient.[21] However, for more complex or variable log formats, a more robust solution is preferable. The

logparser toolkit from the logpai project is a powerful, machine learning-based option that can automatically learn log templates from raw log files and parse them into structured events without needing predefined rules.[23]

- **Implementation Steps:**
  1. Using logparser, define a basic log_format to identify common fields like timestamp and log level. The parser will then analyze the free-text Content portion to identify patterns.
  2. The tool will output a structured representation (e.g., a pandas DataFrame) with columns for each log variable (e.g., Timestamp, LogLevel, MaxStress, ElementID, ConvergenceStatus).
  3. **Structuring for RAG:** The parsed log data will be summarized. Key final results (e.g., max stress, max displacement, pass/fail status) will be added as properties to a "Simulation Run" node in the knowledge graph. This node will be linked to the specific version of the CAD model that was analyzed and the FEA conditions that were used, creating a complete record of the analysis.

1.3. Table 1: Recommended Python Libraries for Engineering Data Ingestion

To aid in the development of the ingestion service, the following table summarizes the recommended Python libraries for each data type, providing a clear rationale for their selection based on the specific demands of this engineering-focused project.

Data Type	Primary Library	Rationale & Key Features	Supporting Libraries	Snippet References
CAD (STEP/IGES)	python-occ	Provides deep geometric kernel access via OpenCASCADE for precise measurements and topological analysis. It is a mature, powerful, and well-established	steputils for high-level parsing of the P21 file structure and extracting header metadata.	[5]

		library for serious CAD work.		
PDF (Tables)	Camelot	Considered state-of-the-art for table extraction from PDFs. It can export directly to pandas DataFrames and supports multiple parsing strategies ('Lattice', 'Stream') to handle various table formats.	pandas for subsequent data manipulation and structuring of the extracted tables.	[15]
PDF (Text/Layout)	pdfplumber	Offers highly detailed information on every character, line, and rectangle, enabling fine-grained control over text extraction and layout analysis, which is superior to more basic libraries.	-	[12]
PDF (Drawings)	VLM (e.g., Florence-2)	Essential for interpreting non-textual information. Can extract structured data (dimensions, tolerances) from complex visual layouts where	Pillow for image handling and pre-processing before sending to the VLM.	[19]

		traditional OCR and parsers fail.		
FEA Logs	logparser (logpai)	A machine learning-based approach that automatically learns log templates from unstructured log files. This is more robust and adaptable than static regex for complex or evolving log formats.	re (Python built-in) for simpler, highly consistent, and fixed-format logs where an ML approach may be overkill.	[21]

Part II: Empowering the Design Agent - A Suite of Generative Engineering Tools

With a robust knowledge core in place, the focus shifts from understanding to action. The DesignAgent will be empowered with a suite of generative engineering tools, enabling it to execute the iterative think → propose → simulate → refine loop. This loop is not a simple, linear sequence of tool calls; it is a stateful optimization process. The output of the simulate tool must directly inform the next input to the propose geometry tool, creating a closed feedback loop. The agent acts as the intelligent controller within this loop, interpreting simulation results and making reasoned decisions about how to modify the design for the next iteration. This elevates the agent from a mere tool user to a genuine design partner. The design of each tool's input and output schema is therefore paramount; they must be machine-readable and contain the necessary information for the agent to reason effectively.

2.1. Architecting the Design Loop in CrewAI

The iterative nature of the design process is well-suited to CrewAI's Hierarchical process model.[26] In this architecture, a "Manager" or "Chief Engineer" agent could oversee the entire workflow, delegating specific tasks—such as proposing an initial design, running a simulation, or refining parameters—to the specialized

DesignAgent. The state of the design, including the current geometry file, the latest simulation report, design parameters, and the iteration count, can be managed by passing the output of one task as the context for the next, ensuring a continuous and informed design evolution.[28]

2.2. Implementing Generative Tools for CrewAI

To ensure consistency, robustness, and traceability, all engineering tools will inherit from a custom base class.

2.2.1. Foundational EngineeringTool Base Class

A custom EngineeringTool class will be created by subclassing CrewAI's BaseTool.[29] This approach provides a standardized structure for all tools and allows for the integration of common functionalities such as:

- **Input Validation:** Using Pydantic's BaseModel for the args_schema to ensure all tool calls have correctly typed and structured inputs.
- **Robust Error Handling:** Implementing try...except blocks within the _run method to gracefully handle failures in the underlying libraries (e.g., a simulation failing to converge).
- **Integrated Logging:** Each tool execution will automatically log its start, end, inputs, and outputs to the central provenance log (detailed in Part IV), creating the Digital Thread.

Python

```python
# src/tools/base_tool.py
from crewai_tools import BaseTool
from pydantic.v1 import BaseModel, Field
import logging
from typing import Type

# Configure a basic logger for the tools
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')

class EngineeringTool(BaseTool):
    """
    A base class for all engineering tools to ensure consistent
    error handling, logging, and input validation.
    """
    name: str = "Base Engineering Tool"
    description: str = "A base class for engineering tools."
    args_schema: Type = BaseModel # Default empty schema

    def _run(self, **kwargs):
        # This method should be overridden by subclasses
        raise NotImplementedError("The _run method must be implemented by a subclass.")

    def run(self, **kwargs):
        """
        Public-facing run method that wraps the internal _run method
        with logging and error handling.
        """
        tool_name = self.name.replace(" ", "")
        logger = logging.getLogger(f"PROVENANCE.{tool_name}")

        try:
            # Validate inputs using the defined Pydantic schema
            validated_args = self.args_schema(**kwargs)
            logger.info(f"Starting execution with args: {validated_args.dict()}")

            # Here, you would integrate with the ProvenanceLogger (see Part IV)
            # provenance_logger.log_tool_start(self.name, validated_args.dict())

            result = self._run(**validated_args.dict())
```

```
            logger.info(f"Successfully completed execution. Result: {result}")
            # provenance_logger.log_tool_end(self.name, result)

            return result
        except Exception as e:
            logger.error(f"Execution failed with error: {e}", exc_info=True)
            # provenance_logger.log_tool_error(self.name, str(e))
            return f"Error in {self.name}: {e}"
```

2.2.2. Tool 1: The Parametric CAD Generator

This tool is responsible for creating precise, manufacturable 3D models from a set of parameters.

- **Function Signature:** generate_parametric_model(params: dict, output_dir: str) -> str
  - **Returns:** The file path to the generated STEP file.
- **Library:** CadQuery is the ideal choice for *generating* parametric geometry in Python. Its fluent, chainable API is more intuitive for scripting models than the lower-level python-occ.[31]
  CadQuery excels at creating models where features are located relative to one another, which is a common paradigm in mechanical design.[33]
- **Implementation:** The tool's function will accept a dictionary of parameters defining the part's dimensions. It will use CadQuery commands to construct a Workplane, draw the base features, and add subsequent features like holes and fillets. The final object is then exported to a STEP file.

Python

```
# src/tools/cad_generator.py
import cadquery as cq
import os
```

```
from pydantic.v1 import BaseModel, Field
from typing import Dict, Any
from.base_tool import EngineeringTool

class ParametricModelInputs(BaseModel):
    params: Dict[str, Any] = Field(description="Dictionary of parameters for the CAD model.")
    output_dir: str = Field(description="Directory to save the output STEP file.")
    filename: str = Field(description="Filename for the output STEP file (without extension).")

class ParametricCADGenerator(EngineeringTool):
    name: str = "Parametric CAD Generator"
    description: str = "Generates a parametric 3D CAD model (STEP file) based on a dictionary of
input parameters."
    args_schema: Type = ParametricModelInputs

    def _run(self, params: Dict[str, Any], output_dir: str, filename: str) -> str:
        """
        Generates a parametric pillow block model.
        Example params: {'length': 80.0, 'height': 60.0, 'thickness': 10.0, 'center_hole_dia': 22.0}
        """
        # Example using the pillow block from CadQuery docs [31]
        result = cq.Workplane("XY").box(
            params.get('length', 80.0),
            params.get('height', 60.0),
            params.get('thickness', 10.0)
        ).faces(">Z").workplane().hole(
            params.get('center_hole_dia', 22.0)
        )

        output_path = os.path.join(output_dir, f"{filename}.step")
        os.makedirs(output_dir, exist_ok=True)

        cq.exporters.export(result, output_path)

        return f"Successfully generated STEP file at: {output_path}"
```

2.2.3. Tool 2: The Topology Optimizer

This tool refines a design by removing material from non-critical areas, resulting in a lightweight yet strong structure.

- **Function Signature:** optimize_topology(geometry_filepath: str, load_conditions: dict, constraints: dict, output_dir: str) -> str
  - **Returns:** The file path to the optimized mesh (e.g., STL).
- **Library:** For classic structural optimization problems, TopOpt [34] and PyTopo3D [35] are excellent choices. They are well-documented Python libraries focused on solving common problems like minimum compliance using established methods.
  DL4TO is a more specialized option if the goal is to integrate deep learning techniques directly into the optimization process.[36] For this implementation, TopOpt is chosen for its clear examples.
- **Implementation:** This tool acts as a wrapper around the chosen library. It will:
  1. Load the input geometry. This may require converting the STEP file into a suitable mesh format first.
  2. Use the load_conditions and constraints dictionaries to programmatically define the optimization problem (e.g., applied forces, fixed points, target volume fraction).
  3. Execute the optimization solver.
  4. Export the resulting density field as an optimized mesh file (e.g., STL).

Python

```
# src/tools/topology_optimizer.py
# Note: This is a conceptual implementation. Actual TopOpt usage will vary.
# See TopOpt documentation for detailed examples.[34]
from.base_tool import EngineeringTool
#... other imports for TopOpt and mesh handling

class TopologyOptimizer(EngineeringTool):
    name: str = "Topology Optimizer"
    description: str = "Optimizes the topology of a given geometry based on load conditions and constraints."
    #... Pydantic args_schema definition...
```

```
    def _run(self, geometry_filepath: str, load_conditions: dict, constraints: dict, output_dir: str) -> str:
        # 1. Load and mesh the input geometry (e.g., from STEP to a voxel grid)
        #    This is a non-trivial step that may require a meshing library.

        # 2. Set up the TopOpt problem
        #    nelx, nely =... # grid dimensions
        #    volfrac = constraints.get('volume_fraction', 0.4)
        #    penal = 3.0
        #    bc =... # Define boundary conditions from load_conditions
        #    problem = ComplianceProblem(bc, penal)

        # 3. Initialize and run the solver
        #    solver = TopOptSolver(problem, volfrac, DensityBasedFilter(nelx, nely, 1.5), gui=None)
        #    optimized_density = solver.optimize()

        # 4. Convert the optimized density field to a mesh (e.g., STL) and save
        #    output_path =...

        return f"Topology optimization complete. Optimized mesh saved to: {output_path}"
```

2.2.4. Tool 3: The Simulation Engine

This tool validates the performance of a design by subjecting it to a virtual physics-based test.

- **Function Signature:** simulate(geometry_filepath: str, material_properties: dict, physics_conditions: dict, output_dir: str) -> str
  - **Returns:** The file path to a structured diagnostic report (JSON).
- **Library:** FEniCS is a powerful, general-purpose FEA framework for solving PDEs, but it has a steep learning curve.[37] For a more accessible yet capable alternative, EasyFEA provides a higher-level, user-friendly Python API for setting up and running simulations.[39]
  PyChrono is another excellent option, particularly for multi-body dynamics and robotics simulations.[40]
  EasyFEA is selected here for its simplicity in demonstrating the concept.
- **Implementation:** The core function of this tool is to produce a machine-readable report that the agent can use for decision-making. It will:

1. Load and mesh the input geometry.
2. Assign material properties (e.g., Young's Modulus, Poisson's Ratio) from the material_properties dictionary.
3. Apply boundary conditions and loads (e.g., fixed faces, applied forces) from the physics_conditions dictionary.
4. Run the FEA solver.
5. **Crucially, post-process the results.** Instead of just generating a visualization, it will extract key metrics like maximum stress, maximum displacement, and solver convergence status, and write them to a structured JSON file. This report is the feedback signal for the agent's refinement loop.

Python

```
# src/tools/simulation_engine.py
import json
import os
from.base_tool import EngineeringTool
#... other imports for EasyFEA and meshing

class SimulationEngine(EngineeringTool):
    name: str = "FEA Simulation Engine"
    description: str = "Performs a Finite Element Analysis (FEA) on a geometry and returns a diagnostic report."
    #... Pydantic args_schema definition...

    def _run(self, geometry_filepath: str, material_properties: dict, physics_conditions: dict, output_dir: str) -> str:
        # 1. Load and mesh the geometry

        # 2. Set up the simulation in EasyFEA
        #    - Define material
        #    - Apply boundary conditions from physics_conditions
        #    - Apply loads

        # 3. Solve the simulation
        #    solver.solve()
```

```
        # 4. Post-process results
        #   max_stress = solver.get_max_stress()
        #   max_displacement = solver.get_max_displacement()
        #   converged = solver.is_converged()

        report = {
            "status": "SUCCESS" if converged else "FAILURE",
            "convergence_details": "Converged successfully." if converged else "Solver did not converge.",
            "results": {
                "max_von_mises_stress_mpa": max_stress,
                "max_displacement_mm": max_displacement,
            },
            "inputs": {
                "geometry_file": geometry_filepath,
                "material": material_properties,
                "physics": physics_conditions
            }
        }

        os.makedirs(output_dir, exist_ok=True)
        report_path = os.path.join(output_dir, "simulation_report.json")
        with open(report_path, 'w') as f:
            json.dump(report, f, indent=4)

        return f"Simulation complete. Diagnostic report saved to: {report_path}"
```

2.3. Integrating Specialized ML Models as Tool "Brains"

The tools described above are primarily for optimizing and validating known design concepts. To introduce true novelty, the agent needs a tool capable of generating entirely new shapes. Generative ML models, such as Variational Autoencoders (VAEs) and Graph Neural Networks (GNNs), excel at this. They learn a compressed "latent space" of plausible shapes from a large dataset. By sampling from this space, they can generate new designs that are statistically similar to the training data but are not simple interpolations.[41] This capability suggests a two-phase design process for the

agent:

Exploration, where it uses the ML model to generate novel starting points, and **Exploitation**, where it uses the iterative simulate-and-refine loop to optimize those starting points.

2.3.1. Architectural Pattern: Model-as-a-Service (MaaS)

Training and running deep learning models for 3D shape generation often requires heavy dependencies like PyTorch, TensorFlow, and specialized libraries like PyTorch Geometric (PyG).[44] To avoid bloating the CrewAI agent's environment and to promote modularity, these models should be wrapped in their own lightweight API service (e.g., using FastAPI). The CrewAI tool then becomes a simple client that sends an HTTP request to this service. This decouples the agent's reasoning environment from the ML inference environment, improving stability, maintainability, and allowing the ML model to be scaled and updated independently.

2.3.2. Optional Tool 4: The Shape Synthesizer

This tool allows the agent to perform the "Exploration" phase of design.

- **Function Signature:** generate_initial_shape(design_prompt: str, design_space_constraints: dict, output_dir: str) -> str
  - **Returns:** The file path to the generated initial mesh (e.g., STL or OBJ).
- **ML Model:** A pre-trained Mesh-VAE [42] or a GNN-based generative model.[45] The model would be trained offline on a large dataset of existing 3D models (e.g., brackets, housings).
- **Implementation:**
  1. The tool's function takes a high-level natural language design_prompt, such as "a lightweight bracket to connect a motor to a frame."
  2. It uses a sentence-transformer model to encode this prompt into a dense vector representation.
  3. This prompt vector is used to sample a corresponding point from the latent space of the pre-trained VAE/GNN.
  4. The model's decoder then generates a 3D mesh from this latent vector.

5. The generated mesh is saved to a file, and its path is returned to the agent, providing a novel starting point for the optimization loop.

2.4. Table 2: Generative Design Tool Implementation Overview

This table defines the clear API contract for each generative tool, outlining its purpose within the agent's workflow and the artifacts it consumes and produces. A well-defined contract is the foundation of an interoperable and effective tool suite, ensuring the agent has the structured information it needs to make intelligent decisions at each step of the design process.

Tool Name	Function Signature	Key Library	Purpose in Design Loop	Output Artifact(s)
ParametricCAD Generator	generate_parametric_model(params: dict,...)	CadQuery	**Propose/Refine:** Creates a precise, manufacturable model based on specific geometric parameters.	STEP file
TopologyOptimizer	optimize_topology(geometry: str,...)	TopOpt	**Refine:** Reduces material usage of a design while respecting structural and performance constraints.	Optimized Mesh (STL)
SimulationEngine	simulate(geometry: str,...)	EasyFEA	**Simulate:** Validates the performance (e.g., stress, displacement) of a design under specified physical conditions.	Diagnostic Report (JSON), Visualization (PNG)

ShapeSynthesizer	generate_initial_shape(prompt: str,...)	Mesh-VAE/GNN	**Explore:** Generates novel, non-obvious design concepts from a high-level prompt to seed the creative process.	Initial Mesh (STL/OBJ)

Part III: System Integration - Configuring the Crew

This section details the integration of the agents and tools within the CrewAI framework. The configuration of the agent's persona and the definition of its tasks are critical for guiding the Large Language Model's (LLM) reasoning and ensuring it can effectively utilize the powerful tools at its disposal.

3.1. Crafting the DesignAgent Persona

The agent's persona—defined by its role, goal, and backstory—is not mere flavor text; it primes the LLM, setting the context for its behavior and decision-making process.[47] A well-crafted persona leads to more focused, expert-level reasoning. The configuration will be stored in

config/agents.yaml.

- **Role:** Senior Computational Design Engineer
  - This is specific and establishes expertise. It's more effective than a generic role like "Designer".[47]
- **Goal:** To autonomously generate, analyze, and optimize a mechanical part that meets the specified performance criteria, design constraints, and manufacturing standards, while exploring novel design possibilities.
  - This goal is outcome-focused and includes quality standards, guiding the agent toward a successful conclusion.[47]
- **Backstory:** You are an expert in generative design, with a PhD in topology

optimization and machine learning. You are proficient in using CAD, FEA, and advanced generative modeling tools. You approach design as an iterative, data-driven process, constantly seeking to improve performance by analyzing simulation results and refining geometric parameters. You can also think "outside the box" by leveraging AI to propose entirely new shapes when conventional methods stagnate.
    - This backstory reinforces expertise, defines a working style, and creates a cohesive persona that aligns with the role and goal.[47]
- **Configuration:**
    - allow_delegation: true: This is crucial. It allows the DesignAgent to delegate information-gathering tasks (e.g., "Find the ISO standard for this material") to the Librarian agent, promoting effective collaboration.[49]
    - verbose: true: Essential during development and debugging to see the agent's thought process and tool usage logs.[49]

config/agents.yaml:

YAML

```yaml
research_librarian:
  role: 'Expert Engineering Librarian'
  goal: 'To find and synthesize technical information from the engineering knowledge base, including CAD models, specifications, FEA logs, and standards, to answer specific queries.'
  backstory: 'You are a meticulous librarian with a degree in information science and a specialization in engineering documentation. You are the keeper of the company''s entire technical history. You excel at navigating complex data repositories, understanding the relationships between different documents, and providing precise, fact-based answers supported by source data.'
  verbose: true

design_agent:
  role: 'Senior Computational Design Engineer'
  goal: 'To autonomously generate, analyze, and optimize a mechanical part that meets specified performance criteria, design constraints, and manufacturing standards, while exploring novel design possibilities.'
  backstory: 'You are an expert in generative design, with a PhD in topology optimization and machine learning. You are proficient in using CAD, FEA, and advanced generative modeling tools. You approach design as an iterative, data-driven process, constantly seeking to improve performance by analyzing simulation results and refining geometric parameters. You can also think "outside the box" by leveraging AI to propose entirely new shapes when conventional methods stagnate.'
```

```
allow_delegation: true
verbose: true
```

3.2. Defining the GenerativeDesignTask

The task definition provides the agent with its specific instructions. A well-defined task breaks down a complex objective into a clear process, specifies the expected output, and leverages context from other tasks.[28] This will be stored in

config/tasks.yaml.

- **Description:** The description must be a detailed, step-by-step guide for the agent. It should not be a "God Task" that is too broad or vague.[47] It will explicitly instruct the agent to follow the
  think -> propose -> simulate -> refine loop.
  - Using the provided design brief in {design_brief}, create a final, optimized 3D model of the part. Your process must be: 1. Consult the Librarian to find any relevant existing designs, specifications, or standards related to the brief. 2. Based on the findings, decide on an initial design. If no suitable starting point exists, use the Shape Synthesizer to propose a novel initial geometry. Otherwise, use an existing design as a baseline. 3. Enter an iterative refinement loop: use the SimulationEngine to analyze the current design's performance. Carefully review the structured diagnostic report. 4. Based on your analysis of the report, decide on modifications to the design's parameters and use the ParametricCADGenerator to create the next iteration. You may also use the TopologyOptimizer to refine the shape. 5. Continue this loop until the performance criteria from the design brief are met or until you determine that no further significant improvement is possible. 6. Your final answer must be the filepath to the final, optimized STEP file and a comprehensive summary report of its performance metrics and the design choices you made.
- **Expected Output:** This clearly defines what "done" looks like for the agent.[27]
  - A markdown string containing two sections. The first section must be the absolute filepath to the final, optimized STEP model. The second section must be a "Final Design Report" summarizing the key performance metrics (e.g., max stress, displacement, final weight) and a brief justification for the final design choices.

- **Context:** The design task will depend on the output of an initial research task performed by the Librarian, demonstrating task dependency.[28]

config/tasks.yaml:

YAML

```yaml
initial_research_task:
  description: 'Review the design brief: "{design_brief}". Search the knowledge base for any existing parts, material specifications (e.g., ISO 10993 compliance), or performance standards that are relevant to this brief. Synthesize your findings into a concise summary.'
  expected_output: 'A summary of relevant information found in the knowledge base, including filepaths to relevant documents and key specifications. If nothing is found, state that clearly.'

generative_design_task:
  description: 'Using the provided design brief "{design_brief}" and the context from the initial research task, execute the full generative design process. Your process must be: 1. Decide on an initial design strategy based on the research context. If no suitable starting point exists, use the Shape Synthesizer to propose a novel initial geometry. Otherwise, start with a parametric model of an existing design. 2. Enter an iterative refinement loop: use the SimulationEngine to analyze the current design. Carefully review the structured diagnostic report. 3. Based on the report, decide on modifications to the design''s parameters and use the ParametricCADGenerator to create the next iteration. You may also use the TopologyOptimizer. 4. Continue this loop until the performance criteria are met or no further improvement is possible. 5. Your final answer must be the filepath to the final, optimized STEP file and a summary report of its performance.'
  expected_output: 'A markdown string containing the absolute filepath to the final STEP model and a "Final Design Report" summarizing key performance metrics and design choices.'
```

3.3. Crew Orchestration and Execution (main.py)

The main.py script is the entry point that brings all the components together. It is responsible for loading the configurations, instantiating the agents and tools, defining the crew, and initiating the workflow.

The script will perform the following steps:

1. **Load Environment Variables:** Load API keys and other configurations from a .env file.

2. **Instantiate Tools:** Create instances of all the custom tools: RAGTool (for the Librarian), ParametricCADGenerator, TopologyOptimizer, SimulationEngine, and ShapeSynthesizer.
3. **Instantiate Agents:** Create instances of the research_librarian and design_agent using the Agent class, loading their configuration from agents.yaml and passing the instantiated tools to the tools parameter.[49]
4. **Instantiate Tasks:** Create instances of the initial_research_task and generative_design_task, assigning the appropriate agent to each and setting the context for the design task to depend on the research task.
5. **Assemble the Crew:** Create the Crew object, passing in the list of agents and tasks. The process will be set to Process.sequential for this example to ensure the research happens before the design, but a Process.hierarchical setup would also be highly effective, allowing a manager agent to coordinate the two specialists.[27]
6. **Kickoff:** Call the crew.kickoff() method, passing the initial design_brief as an input variable, which will populate the {design_brief} placeholders in the task descriptions.[50]

src/main.py:

Python

```
import os
from dotenv import load_dotenv
from crewai import Agent, Task, Crew, Process

# Load environment variables
load_dotenv()

# Import your custom tools
# from tools.rag_tool import RAGTool
from tools.cad_generator import ParametricCADGenerator
from tools.topology_optimizer import TopologyOptimizer
from tools.simulation_engine import SimulationEngine
from tools.shape_synthesizer import ShapeSynthesizer

def run_design_crew():
    # --- 1. Instantiate Tools ---
```

```
    # rag_tool = RAGTool() # Assuming a RAG tool for the librarian
    cad_tool = ParametricCADGenerator()
    topo_opt_tool = TopologyOptimizer()
    sim_tool = SimulationEngine()
    shape_gen_tool = ShapeSynthesizer()

    # --- 2. Define Agents ---
    # Note: In a real implementation, you'd load from agents.yaml
    research_librarian = Agent(
        role='Expert Engineering Librarian',
        goal='To find and synthesize technical information from the engineering knowledge base...',
        backstory='You are a meticulous librarian...',
        # tools=[rag_tool],
        verbose=True
    )

    design_agent = Agent(
        role='Senior Computational Design Engineer',
        goal='To autonomously generate, analyze, and optimize a mechanical part...',
        backstory='You are an expert in generative design...',
        tools=[cad_tool, topo_opt_tool, sim_tool, shape_gen_tool],
        allow_delegation=True,
        verbose=True
    )

    # --- 3. Define Tasks ---
    # Note: In a real implementation, you'd load from tasks.yaml
    initial_research_task = Task(
        description='Review the design brief: "{design_brief}". Search the knowledge base...',
        expected_output='A summary of relevant information found...',
        agent=research_librarian
    )

    generative_design_task = Task(
        description='Using the provided design brief "{design_brief}" and the context from the initial research task, execute the full generative design process...',
        expected_output='A markdown string containing the absolute filepath to the final STEP model and a "Final Design Report"...',
        agent=design_agent,
```

```
        context=[initial_research_task] # This task uses the output of the research task
    )

    # --- 4. Assemble and Run the Crew ---
    design_crew = Crew(
        agents=[research_librarian, design_agent],
        tasks=[initial_research_task, generative_design_task],
        process=Process.sequential,
        verbose=2 # Set to 1 or 2 for detailed logging
    )

    # --- 5. Kickoff the process ---
    design_brief_input = {
        'design_brief': 'Design a lightweight mounting bracket for a NEMA 17 stepper motor. It must withstand a static load of 50N applied to the motor face without yielding. Material should be 6061-T6 Aluminum. Find relevant material properties and standards.'
    }

    result = design_crew.kickoff(inputs=design_brief_input)
    print("\n\n########################")
    print("## Crew Execution Result:")
    print("########################")
    print(result)

if __name__ == "__main__":
    run_design_crew()
```

Part IV: Achieving End-to-End Traceability and Data Provenance

In any serious engineering or production-grade AI workflow, simply achieving the correct output is not enough. The system must be auditable, debuggable, and reproducible. This requires a robust traceability framework. Standard application logging is passive and often incomplete. Drawing from MLOps best practices and the principles of AI accountability, the proposed system will implement an active, structured "Digital Thread".[52] This is not merely a log of events, but a verifiable,

versioned record of the entire causal chain of actions and decisions made by the agents. Each entry in this thread will contain enough information to re-execute that specific step, providing unparalleled transparency and control.[55]

4.1. An MLOps Foundation for Agentic Workflows

The principles of MLOps, traditionally applied to model training and deployment, are directly applicable to managing the lifecycle of this agentic system.

- **Versioning:** Every component of the system must be under version control. This includes not only the agent and tool source code (managed with Git) but also the data and artifacts. Datasets ingested into the knowledge base, agent configurations (agents.yaml, tasks.yaml), and every model artifact generated during a run (STEP files, simulation reports) should be versioned using a tool like DVC (Data Version Control).[53] This ensures that any given result can be traced back to the exact code, data, and configuration that produced it.
- **Automation (CI/CD):** The entire crew.kickoff() process should be wrapped in an automated pipeline (e.g., using GitHub Actions, Jenkins). A new design request could trigger this pipeline, which would execute the crew, run automated tests on the output, and store the versioned results and provenance log in a central repository.[53] This removes manual steps, reduces error, and ensures a consistent execution environment.
- **Monitoring:** While a full implementation is beyond this scope, a production system would monitor key performance indicators of the agentic workflow. This includes tool success and failure rates, LLM costs, task completion times, and the quality of the final designs.[59] Dashboards (e.g., using Grafana) can provide visibility into the health and efficiency of the system.

4.2. Implementing the Digital Thread: A Provenance Log

The core of the traceability system is a structured, queryable provenance log. This log will be implemented as a dedicated ProvenanceLogger class that is integrated into the EngineeringTool base class and the agent execution loop. It will capture every significant event in the agent's workflow.

The log will be structured as a directed acyclic graph (DAG), where each log entry has a unique ID and can reference the ID(s) of its parent event(s). This explicitly models the flow of data and logic. For example, a TOOL_END event for the SimulationEngine would have the TOOL_START event as its parent. The subsequent THOUGHT event from the agent analyzing the result would, in turn, have that TOOL_END event as its parent. This creates an unbroken chain of causality.

The logger will write structured JSON entries to a file, which can be easily ingested into a logging platform like the ELK Stack or a dedicated database for analysis and auditing.[58]

Python

```python
# src/utils/provenance_logger.py
import json
import uuid
import datetime
import hashlib
import os

class ProvenanceLogger:
    def __init__(self, log_file_path: str):
        self.log_file_path = log_file_path
        os.makedirs(os.path.dirname(log_file_path), exist_ok=True)

    def _hash_file(self, file_path: str) -> str:
        """Computes the SHA256 hash of a file."""
        if not os.path.exists(file_path):
            return None
        h = hashlib.sha256()
        with open(file_path, 'rb') as f:
            while True:
                chunk = f.read(h.block_size)
                if not chunk:
                    break
                h.update(chunk)
        return f"sha256:{h.hexdigest()}"
```

```
    def log_event(self, event_data: dict):
        """Writes a structured event to the log file."""
        log_entry = {
            "entry_id": str(uuid.uuid4()),
            "timestamp": datetime.datetime.now(datetime.timezone.utc).isoformat(),
            **event_data
        }
        with open(self.log_file_path, 'a') as f:
            f.write(json.dumps(log_entry) + '\n')
        return log_entry['entry_id']

    def log_tool_run(self, agent_id, tool_name, tool_inputs, tool_output, input_artifacts, output_artifacts,
llm_trace, code_version, parent_ids=):
        """A helper to log a complete tool execution cycle."""

        # Hash input and output artifacts
        hashed_inputs = {fp: self._hash_file(fp) for fp in input_artifacts}
        hashed_outputs = {fp: self._hash_file(fp) for fp in output_artifacts}

        event_data = {
            "parent_ids": parent_ids,
            "agent_id": agent_id,
            "event_type": "TOOL_EXECUTION",
            "details": {
                "tool_name": tool_name,
                "tool_inputs": tool_inputs,
                "tool_output": tool_output,
                "input_artifacts": hashed_inputs,
                "output_artifacts": hashed_outputs,
                "llm_trace": llm_trace,
                "code_version": code_version
            }
        }
        return self.log_event(event_data)
```

This logger would be instantiated in main.py and passed to the tools and agents to be used throughout the execution.

4.3. Table 3: Schema for the Digital Provenance Log

This schema defines the structure of each entry in the provenance log. Its comprehensive nature is designed to capture all information necessary for full reproducibility and auditability, transforming a simple log into a powerful accountability tool.[55]

Field Name	Data Type	Description	Example
entry_id	UUID	A unique identifier for this specific log entry.	f47ac10b-58cc-4372-a567-0e02b2c3d479
parent_ids	List	A list of entry_ids for the parent events that triggered this one, forming the causal chain.	['e2a1b0c9-4b1f-4a5e-8b1a-2b0c1d2e3f4a']
timestamp	ISO 8601	The precise UTC timestamp when the event occurred.	2025-07-04T10:30:00.123456Z
agent_id	String	The name or identifier of the agent that performed the action (e.g., design_agent).	design_agent
event_type	Enum	The type of event, e.g., TOOL_START, TOOL_END, THOUGHT, DELEGATION, FINAL_ANSWER.	TOOL_END
tool_name	String	The name of the tool that was executed, if applicable.	SimulationEngine

tool_inputs	JSON/Dict	The exact, serializable dictionary of arguments passed to the tool.	{'geometry_filepath': './artifacts/v2.step', 'physics_conditions': {...}}
tool_output	JSON/Dict	The structured, serializable output returned by the tool (e.g., a summary or status).	{'max_stress': 350.5, 'status': 'SUCCESS'}
input_artifacts	Dict	A dictionary mapping input file paths to their cryptographic hashes (e.g., SHA256).	{'./artifacts/v2.step': 'sha256:abc...'}
output_artifacts	Dict	A dictionary mapping output file paths to their cryptographic hashes.	{'./reports/v2_sim.json': 'sha256:def...'}
llm_trace	JSON/Dict	The raw prompt sent to the LLM, the model's "thought" process, and the final response for this step.	{'thought': 'The stress is too high...', 'raw_response': '...'}
code_version	String	The Git commit hash of the agent/tool codebase at the time of execution.	a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0

4.4. A Walkthrough of the Provenance Trail

To illustrate the power of this integrated system, consider a full design cycle initiated by the prompt: "Design a lightweight bracket." The Digital Thread would capture the following chain of events, each as a structured log entry linked by parent_ids:

1. **Crew Kickoff:** An initial entry logs the start of the process, the input prompt, and the code version.
2. **Librarian Research:** The DesignAgent delegates a task to the Librarian. A DELEGATION event is logged. The Librarian uses its RAGTool, and a TOOL_EXECUTION event is logged, capturing the query ("search for bracket standards") and the retrieved documents. The output summary becomes the context for the next step.
3. **Exploration (Shape Synthesis):** The DesignAgent's THOUGHT process is logged: "No suitable starting design found. I will generate a novel shape." It then calls the ShapeSynthesizer. A TOOL_EXECUTION entry is created, logging the prompt, the output_artifacts (the hash of initial_shape_v1.stl), and the code_version.
4. **Simulation (Iteration 1):** The agent's next THOUGHT is logged: "Now I will simulate the initial shape." It calls the SimulationEngine. A new TOOL_EXECUTION entry is created. Its parent_id points to the ShapeSynthesizer entry. Its input_artifacts contains the hash of initial_shape_v1.stl, and its output_artifacts contains the hash of sim_report_v1.json. The tool_output field contains the structured results: {max_stress: 500,...}.
5. **Refinement (Iteration 1):** The agent's THOUGHT is logged: "Stress is 500 MPa, which is too high. I will increase the thickness of the main flange." It calls the ParametricCADGenerator with new parameters. A TOOL_EXECUTION entry is logged, with its parent being the simulation entry. It captures the new parameters {'thickness': 12.0} and the output artifact hash for refined_shape_v2.step.
6. **Loop and Conclude:** This cycle of Simulate -> Think -> Refine continues. Each step creates a new, linked entry in the log. Finally, the agent produces its FINAL_ANSWER, which is logged with a parent ID pointing to the last successful simulation.

An auditor or engineer can now pick the final optimized_bracket_v5.step file, look up its hash in the provenance log, and trace its entire history backward. They can see every decision the agent made, every simulation result it reviewed, every tool it used with the exact inputs, and the specific version of the code that was running at the time. This creates a fully transparent, auditable, and reproducible record of the autonomous design process, which is the ultimate goal of a production-ready agentic engineering system.

Works cited

1. NirDiamant/Controllable-RAG-Agent: This repository provides an advanced Retrieval-Augmented Generation (RAG) solution for complex question answering.

It uses sophisticated graph based algorithm to handle the tasks. - GitHub, accessed July 11, 2025, https://github.com/NirDiamant/Controllable-RAG-Agent
2. mytechnotalent/Simple-RAG-Agent - GitHub, accessed July 11, 2025, https://github.com/mytechnotalent/Simple-RAG-Agent
3. RAGFlow is an open-source RAG (Retrieval-Augmented Generation) engine based on deep document understanding. - GitHub, accessed July 11, 2025, https://github.com/infiniflow/ragflow
4. Agentic RAG - GitHub Pages, accessed July 11, 2025, https://langchain-ai.github.io/langgraph/tutorials/rag/langgraph_agentic_rag/
5. steputils - PyPI, accessed July 11, 2025, https://pypi.org/project/steputils/
6. STEPutils is a Python package to manage STEP model data. - GitHub, accessed July 11, 2025, https://github.com/mozman/steputils
7. pythonOCC | Open CASCADE Technology, accessed July 11, 2025, https://dev.opencascade.org/project/pythonocc
8. p21 - STEP-file - STEPutils 0.1a2 documentation - Read the Docs, accessed July 11, 2025, https://steputils.readthedocs.io/en/latest/p21.html
9. [2505.13535] Information Extraction from Visually Rich Documents using LLM-based Organization of Documents into Independent Textual Segments - arXiv, accessed July 11, 2025, https://arxiv.org/abs/2505.13535
10. [2502.18179] Problem Solved? Information Extraction Design Space for Layout-Rich Documents using LLMs - arXiv, accessed July 11, 2025, https://arxiv.org/abs/2502.18179
11. Document Parsing Unveiled: Techniques, Challenges, and Prospects for Structured Information Extraction - arXiv, accessed July 11, 2025, https://arxiv.org/html/2410.21169v2?ref=chitika.com
12. jsvine/pdfplumber: Plumb a PDF for detailed information about each char, rectangle, line, et cetera — and easily extract text and tables. - GitHub, accessed July 11, 2025, https://github.com/jsvine/pdfplumber
13. pdfplumber-aemc - PyPI, accessed July 11, 2025, https://pypi.org/project/pdfplumber-aemc/0.5.28/
14. Python, Open-Source Libraries for Efficient PDF Management - DZone, accessed July 11, 2025, https://dzone.com/articles/python-open-source-libraries-pdf-management
15. Camelot: PDF Table Extraction for Humans — Camelot 1.0.0 documentation, accessed July 11, 2025, https://camelot-py.readthedocs.io/
16. How It Works — Camelot 1.0.0 documentation - Read the Docs, accessed July 11, 2025, https://camelot-py.readthedocs.io/en/master/user/how-it-works.html
17. Camelot Documentation, accessed July 11, 2025, https://camelot-py.readthedocs.io/_/downloads/en/stable/pdf/
18. Quickstart — Camelot 1.0.0 documentation - Read the Docs, accessed July 11, 2025, https://camelot-py.readthedocs.io/en/master/user/quickstart.html
19. [2505.01530] Automated Parsing of Engineering Drawings for Structured Information Extraction Using a Fine-tuned Document Understanding Transformer - arXiv, accessed July 11, 2025, https://arxiv.org/abs/2505.01530
20. [2411.03707] Fine-Tuning Vision-Language Model for Automated Engineering

Drawing Information Extraction - arXiv, accessed July 11, 2025, https://arxiv.org/abs/2411.03707
21. Parse and Clean Log Files in Python - GeeksforGeeks, accessed July 11, 2025, https://www.geeksforgeeks.org/python/parse-and-clean-log-files-in-python/
22. How To Use Python To Parse Server Log Files - GeeksforGeeks, accessed July 11, 2025, https://www.geeksforgeeks.org/python/how-to-use-python-to-parse-server-log-files/
23. logpai/logparser: A machine learning toolkit for log parsing [ICSE'19, DSN'16] - GitHub, accessed July 11, 2025, https://github.com/logpai/logparser
24. Top Python libraries for text extraction from PDFs - AZ Big Media, accessed July 11, 2025, https://azbigmedia.com/business/top-python-libraries-for-text-extraction-from-pdfs/
25. PDFPlumber - Python LangChain, accessed July 11, 2025, https://python.langchain.com/docs/integrations/document_loaders/pdfplumber/
26. CrewAI: Introduction, accessed July 11, 2025, https://docs.crewai.com/en/introduction
27. Introduction to CrewAI Agents, Tasks, and Crews | CodeSignal Learn, accessed July 11, 2025, https://codesignal.com/learn/courses/getting-started-with-crewai-agents-and-tasks/lessons/introduction-to-agents-tasks-and-crews-in-crewai
28. Tasks - CrewAI Documentation, accessed July 11, 2025, https://docs.crewai.com/concepts/tasks
29. Tools - CrewAI, accessed July 11, 2025, https://docs.crewai.com/en/concepts/tools
30. Please suggest a tutorial to create a custom tool in crew.ai : r/crewai - Reddit, accessed July 11, 2025, https://www.reddit.com/r/crewai/comments/1innni1/please_suggest_a_tutorial_to_create_a_custom_tool/
31. Introduction — CadQuery Documentation - Pythonhosted.org, accessed July 11, 2025, https://pythonhosted.org/cadquery/intro.html
32. CadQuery 2 Documentation — CadQuery Documentation, accessed July 11, 2025, https://cadquery.readthedocs.io/en/latest/
33. CadQuery Examples - Pythonhosted.org, accessed July 11, 2025, https://pythonhosted.org/cadquery/examples.html
34. TopOpt — Topology Optimization in Python — TopOpt 0.0.1-alpha.1 documentation, accessed July 11, 2025, https://pytopopt.readthedocs.io/en/latest/
35. PyTopo3D: A Python Framework for 3D SIMP-based Topology Optimization - arXiv, accessed July 11, 2025, https://arxiv.org/html/2504.05604v1
36. DL4TO is a Python library for 3D topology optimization that is based on PyTorch and allows easy integration with neural networks. - GitHub, accessed July 11, 2025, https://github.com/dl4to/dl4to
37. FEniCS | FEniCS Project, accessed July 11, 2025, https://fenicsproject.org/

38. Python FEA Simulations with FEniCS and FEATool Multiphysics - Medium, accessed July 11, 2025, https://medium.com/multiphysics/multiphysics-simulations-in-python-with-fenics-and-featool-310c775e5fdc
39. EasyFEA is a user-friendly Python library that simplifies finite element analysis. - GitHub, accessed July 11, 2025, https://github.com/matnoel/EasyFEA
40. PyChrono - An Open-Source Physics Engine - Project Chrono, accessed July 11, 2025, https://projectchrono.org/pychrono/
41. IGLICT/MeshPooling: Code for 'Mesh Variational Autoencoders with Edge Contraction Pooling' - GitHub, accessed July 11, 2025, https://github.com/IGLICT/MeshPooling
42. Interpretable cardiac anatomy modeling using variational mesh autoencoders - PMC, accessed July 11, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC9813669/
43. Using a Deep Generative Model to Generate and Manipulate 3D Object Representation - DiVA, accessed July 11, 2025, https://kth.diva-portal.org/smash/get/diva2:1837778/FULLTEXT01.pdf
44. pyg-team/pytorch_geometric: Graph Neural Network Library for PyTorch - GitHub, accessed July 11, 2025, https://github.com/pyg-team/pytorch_geometric
45. InfoGNN: End-to-end deep learning on mesh via graph neural networks - arXiv, accessed July 11, 2025, https://arxiv.org/html/2503.02414v1
46. Graph Neural Networks for 3D Shape Analysis - Drexel University, accessed July 11, 2025, https://researchdiscovery.drexel.edu/esploro/outputs/doctoral/Graph-Neural-Networks-for-3D-Shape/991021890111904721
47. Crafting Effective Agents - CrewAI Documentation, accessed July 11, 2025, https://docs.crewai.com/guides/agents/crafting-effective-agents
48. Agentic Workflows Using CrewAI - Medium, accessed July 11, 2025, https://medium.com/@StevieLKim/crewai-and-me-part-1-2fa579bcb07c
49. Customize Agents - CrewAI Documentation, accessed July 11, 2025, https://docs.crewai.com/learn/customizing-agents
50. Agents - CrewAI Documentation, accessed July 11, 2025, https://docs.crewai.com/concepts/agents
51. docs.crewai.com, accessed July 11, 2025, https://docs.crewai.com/concepts/tasks#:~:text=In%20the%20CrewAI%20framework%2C%20a,wide%20range%20of%20action%20complexities.
52. MLOps best practices | Intel® Tiber™ AI Studio - Cnvrg.io, accessed July 11, 2025, https://cnvrg.io/mlops-best-practices/
53. 10 MLOps Best Practices Every Team Should Be Using - Mission Cloud Services, accessed July 11, 2025, https://www.missioncloud.com/blog/10-mlops-best-practices-every-team-should-be-using
54. MLOps best practices - Harness Developer Hub, accessed July 11, 2025, https://developer.harness.io/docs/continuous-integration/development-guides/mlops/mlops-best-practices/
55. AI Output Disclosures: Use, Provenance, Adverse Incidents, accessed July 11,

2025, https://www.ntia.gov/issues/artificial-intelligence/ai-accountability-policy-report/developing-accountability-inputs-a-deeper-dive/information-flow/ai-output-disclosures
56. Data Provenance – Explanation & Overview - SnapLogic, accessed July 11, 2025, https://www.snaplogic.com/glossary/data-provenance
57. AI Agents and Why Storage is the Key to Smarter, More Reliable Agents, accessed July 11, 2025, https://fil.org/blog/ai-agents-and-why-storage-is-the-key-to-smarter-more-reliable-agents
58. MLOps Best Practices and How to Apply Them - Signity Software Solutions, accessed July 11, 2025, https://www.signitysolutions.com/tech-insights/mlops-best-practices
59. From Prototype to Production: MLOps Best Practices Using Runpod's Platform, accessed July 11, 2025, https://www.runpod.io/articles/guides/mlops-best-practices
60. The MLOps Playbook: 6 Best Practices for Success in 2025 - Instatus blog, accessed July 11, 2025, https://instatus.com/blog/mlops-playbook