Incorporating X-ray Microscopy into Self-Driving Laboratories

Gordon Research Conference 2026: AI for Materials, Energy, and Chemical Sciences

Nathan S. Johnson
Carl Zeiss Research Microscopy Solutions

1 Introduction

Self-driving laboratories have emerged as a powerful paradigm for accelerating scientific discovery by tightly coupling experiment execution, data analysis, and decision-making in automated closed-loop workflows. While substantial progress has been made in automating synthesis, simulation, and data-driven optimization, advanced microscopy remains a persistent bottleneck. In particular, X-ray microscopy requires expert knowledge to safely configure instrument geometry, select acquisition parameters, center samples, and interpret imaging results. These tasks are traditionally performed manually and are difficult to encode using fixed-rule automation alone.

This document presents a deeply technical description of a multi-agent large-language-model (LLM) control system developed to autonomously operate a ZEISS Versa 630 X-ray microscope. The system is designed as a modular microscopy control layer that can be integrated into broader self-driving laboratory frameworks. Unlike monolithic LLM control approaches, the system emphasizes explicit orchestration, deterministic hardware interfaces, structured state, and layered memory, enabling safe, repeatable, and extensible autonomous microscopy.

2 Software Architecture Overview

The system is implemented in Python and organized around four tightly integrated subsystems: (1) a LangGraph-based multi-agent controller, (2) MCP-exposed deterministic hardware and geometry services, (3) a layered retrieval-augmented generation (RAG) memory system, and (4) a metrics, logging, and profiling framework. The architecture separates reasoning, execution, and state management, which is essential for safe autonomy on physical laboratory equipment.

At runtime, an interactive driver initializes an isolated workspace, constructs an initial `AgentState`, and executes a compiled LangGraph application until the workflow terminates. All intermediate artifacts, metadata sidecars, and logs are written to a structured directory hierarchy, enabling reproducibility and post hoc inspection.

Major Python packages include `langgraph` (state-machine orchestration), `langchain` (model abstraction and message types), `fastmcp` (MCP servers and typed tool schemas), `chromadb/langchain_chroma` (vector stores), and scientific Python libraries (NumPy, OpenCV/scikit-image where applicable) for image processing. Instrument control relies on the ZEISS XradiaPy API, wrapped behind MCP tools to ensure deterministic, auditable behavior.

3 Agent State Model

All nodes in the graph operate over a shared mutable `AgentState` dictionary. This state is the authoritative working memory of the system and is designed to minimize dependence on long conversational history. Table 1 summarizes the major fields.

State key	Type	Description / semantics
`messages`	list[Message]	Bounded transcript (Human/AI/Tool messages). Older messages may be pruned/summarized to enforce token and count budgets.
`world_model`	dict	Authoritative structured working memory (e.g., `sample`, `instrument`, `safety`, `session`). Injected into every agent prompt.
`tasks`	list[dict]	Task list. Each task typically includes `task_id`, `description`, `assigned_to`, `status`, and optional `result`.
`assigned_tasks_for_agent`	list[str]	Current batch of task IDs assigned to an agent. Used to reduce LLM overhead via batched execution.
`last_tool_span`	dict	Captured tool span (AI tool-call message + ToolMessages) for post-processing by world update.
`task_trace`	dict	Per-task structured trace (tool calls, results, timestamps, verifier decisions). Enables auditability and debugging.
`artifacts_index`	dict	Registry of generated files (images, sidecars, recon outputs) with metadata linking artifacts to tasks and sample/session context.
`lessons`	list[dict]	Short-term experiential memory distilled by world-update (e.g., "what worked" / "what failed"). Capped and injected into prompts.
`memory_context`	str	Retry hints / additional context injected on failure (e.g., last error summary, suggested correction).
`last_error`	str	Most recent detected error string (tool failure or exception). Used by verifier and supervisor.
`retry_count`	int	Global or per-workflow retry counter; incremented by verifier on failure.
`max_retries`	int	Maximum retries before escalation/abort (also configurable globally).
`next`	str	Routing directive used by LangGraph conditional edges (e.g., `instrument`, `image`, `world_update`, `supervisor`, `FINISH`).
`decision_log`	list[dict]	Timestamped log of supervisor routing and rationale for traceability.

Table 1: Representative `AgentState` schema used by the LangGraph controller. Exact key set may include additional workflow-specific fields (e.g., workspace paths, metrics handles).

4 LangGraph Topology and Agent Roles

The LangGraph application consists of five nodes: **Supervisor**, **Instrument Agent**, **Image Agent**, **World Update**, and **Verifier**. The supervisor performs task planning and batch assignment. The instrument agent executes microscope control actions via MCP (motion, objective/filter selection, acquisition). The image agent executes analysis tasks (segmentation, centroid extraction, transmission computation) and geometry reasoning via deterministic geometry tools.

After agent tool execution, the world-update node reduces the raw tool span into a structured patch, merging updates into `world_model`, indexing any produced artifacts, and appending task trace entries. The verifier then evaluates whether the current task is complete, failed, or requires

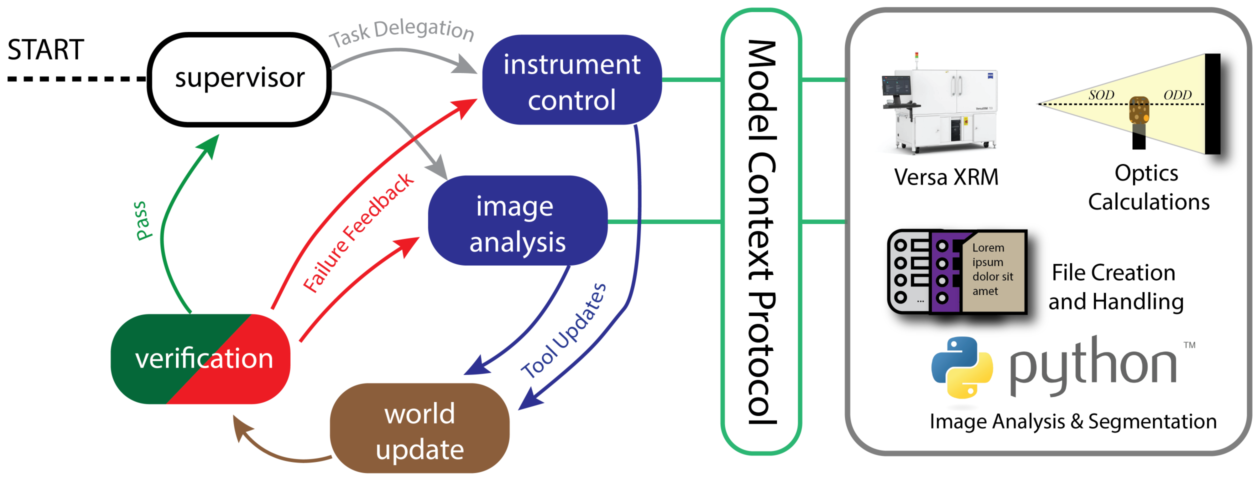

Figure 1: High-level control flow of the LangGraph application. The supervisor plans and assigns task batches. Instrument and image agents execute MCP tool calls, producing a tool span. A world-update node reduces tool outputs into a structured patch (world model, artifacts, trace, lessons). A verifier node evaluates completion, handles retries, and routes control to the next node via `state['next']`.

retry, and sets `state['next']` accordingly.

A critical performance optimization is **batch execution**: the supervisor assigns a list of task IDs to an agent, and the agent is instructed to execute all tasks in sequence. Batch continuation is implemented by verifier routing back to the same agent (agent → world update → verifier → agent) until the batch is exhausted, rather than forcing a supervisor intervention after each tool call.

Figure 1 summarizes the primary LangGraph control flow. The key design pattern is that execution proceeds *agent* → *world update* → *verifier*, with the verifier routing back to the same agent for batch continuation, retry, or escalation to the supervisor for replanning.

5 Formal Pseudocode for the Agent Loop

Algorithm 1 formalizes the control loop implemented by the LangGraph state machine. The pseudocode focuses on the design pattern in which (i) the supervisor emits tasks and assignments, (ii) agents execute tools via MCP, (iii) tool outputs are reduced into structured state patches, and (iv) a verifier gate controls retries and routing.

6 Deterministic Tool Execution via MCP

A central safety and robustness feature is that the LLM agents never directly manipulate the microscope workstation or call vendor APIs in-process. Instead, all actuation and analysis capabilities are exposed as a curated, typed "tool surface" via the Model Context Protocol (MCP). In this system, the LangGraph controller is an MCP *client*: it issues JSON-RPC tool calls and receives structured responses. Each MCP server is a small, single-responsibility Python service implemented with `fastmcp`.

This separation yields two practical guarantees. First, hardware interactions are deterministic and auditable: tool outputs are machine-readable dictionaries rather than free-form text, and each call can be logged with its arguments and results. Second, safety policies can be enforced at the interface boundary: tools validate inputs, can clamp to calibrated limits, and can refuse unsafe requests without relying on the LLM to infer hidden constraints. In practice, this means that failures are dominated by planning quality (task decomposition and sequencing) rather than by unpredictable actuation behavior.

6.1 MCP mechanics and run context

The tool runner in the LangGraph application (e.g., a `ToolNode`) injects a per-run `working_directory` into tool calls so that each server reads and writes only within the current run workspace. Servers follow a sidecar-first philosophy: whenever a tool produces an artifact (TIFF image, recipe file, histogram figure, geometry report), it also writes a compact JSON sidecar capturing parameters and state required for reproducibility and downstream tools (e.g., pixel size, image shape, stage pose, source/detector positions, objective/filter selection, voltage/power, binning, and exposure time).

6.2 Versa MCP server (`versa_server.py`)

The `Versa` server is the primary hardware interface. It wraps the ZEISS vendor API (`XradiaPy.Core.XRM`) behind deterministic tools for instrument snapshotting, X-ray source control, axis motion, and image acquisition. Internally it uses standard libraries (`threading`, `time`, `uuid`, `os`/`sys`) plus `anyio` and `fastmcp`. Every acquisition writes both an image file and a JSON metadata sidecar (via a dedicated helper), which becomes the authoritative record consumed by the geometry and analysis servers.

Operationally, the Versa server is intentionally conservative: it provides small, composable primitives ("query current positions", "move axis", "acquire image") rather than a monolithic "run the workflow" command. This keeps higher-level sequencing in the LangGraph controller, where retries, verification steps, and cross-agent coordination can be expressed explicitly and logged as part of the agent state.

6.3 Geometry MCP server (`geometry_server.py`)

The `Geometry` server provides deterministic, calibration-aware computations that should not be re-derived by the LLM at runtime. It is implemented using `sqlite3` (for calibration/limits lookup), `dataclasses` (for typed limit records), and basic numerics (`math`). The server loads global limits and detector-specific constraints from a bundled SQLite database (`instrument_limits/opt/limits.sqlite`) and can emit JSON reports that record both the selected configuration and the limits used.

The server implements two main classes of functionality. First, it exposes sidecar-based centering transforms (`recenter_xy_from_sidecar` and `recenter_yz_from_sidecar`) that convert an image-space centroid (row/column in pixels) into a stage-space correction in microns. These tools read pixel size, image shape, stage pose, and θ from the acquisition sidecar, validate that θ is consistent with the intended view (e.g., XY expects $\theta \approx 0^\circ$), and then apply fixed sign conventions to map image rows/columns into stage axes. The output includes both the delta movement and the proposed new stage position.

Second, the server exposes geometry-selection utilities that choose source-to-object and object-to-detector distances to meet imaging requirements while respecting calibrated mechanical and detector limits. Given a sample bounding box and a desired pixel size (or field of view), these tools search allowable detector modes and binning options and return a feasible configuration (including recommended source/detector Z positions), or a "no-solution" report that explains which constraints cannot be satisfied.

6.4 Image analysis MCP server (`image_analysis_server.py`)

The `ImageAnalysis` server exposes deterministic image processing primitives used in centering, transmission correction, and workflow verification. It is built on `numpy`, `Pillow` (TIFF I/O), `scikit-image` (thresholding and region measurements), and `matplotlib` (figure generation). A dedicated TIFF loader handles common microscopy edge cases (multi-page TIFFs, RGB(A) encodings, and volumetric stacks) and returns both the image array and basic metadata.

At the tool level, the server provides Otsu-threshold segmentation and connected-component measurements (area, centroid, bounding box) via `skimage.filters` and `skimage.measure`. It also provides histogram plotting and annotation utilities (e.g., overlaying a centroid marker and bounding box) to create human-auditable artifacts. These tools are intentionally low-level and deterministic: the agent decides *when* to segment and *how* to use the resulting measurements, but the pixel-level operations themselves are reproducible and parameterized.

6.5 Recipe MCP server (`recipe_server.py`)

The `Versa-Recipe` server wraps the ZEISS recipe API (`XradiaPy.Recipe.Recipe`) to enable programmatic generation and execution of acquisition recipes. Tools in this server modify key recipe parameters (motion, optics, and acquisition settings), write the updated recipe to the run workspace, and emit JSON snapshots of recipe state for traceability (e.g., objective, filter, start/end angle, number of points, and selected acquisition parameters). This supports tomography and other multi-projection acquisitions where configuration is naturally represented as a recipe file rather than a sequence of individual "single-image" commands.

6.6 File handling MCP server (`file_handling_server.py`)

The `FileReading` server provides guarded file I/O utilities required for autonomous workflows: listing directory contents, reading JSON and text files, validating existence, and optionally searching within subdirectories. The server resolves all non-absolute paths relative to `working_directory`, expands user-home shortcuts, and returns structured error messages rather than raw tracebacks. Centralizing these operations reduces the chance of path bugs in LLM-generated calls and enables consistent error handling across agents.

6.7 Python execution MCP server (`python_execution_server.py`)

Finally, the `python_executor` server is an "escape hatch" for analysis tasks outside the curated tool surface. It accepts a Python source string, optionally writes it to the run `artifacts/` directory for reproducibility, and executes it in a subprocess while capturing stdout/stderr and wall-clock execution time. The layered memory and supervisor policies explicitly prefer dedicated tools (segmentation, geometry, file I/O) over free-form code for safety and repeatability; free-form execution is reserved for well-scoped computations and figure generation that would otherwise require expanding the MCP surface area.

7 Layered Retrieval-Augmented Memory

The memory subsystem is layered in practice: (i) a structured world model (authoritative working memory), (ii) bounded "lessons" distilled from recent tool spans, and (iii) RAG over episodic and contextual vector stores. Episodic memory is agent-specific and optimized for procedural recall; contextual memory is shared and gated for analysis-heavy queries. Retrieval hyperparameters (top-k, keyword gates, embedding model selection) are explicit and logged, allowing reproducible tuning.

8 Hyperparameters and Operational Knobs

Key operational hyperparameters include the model/provider choice, LangGraph recursion limits, batching policy, maximum retries, transcript retention limits, pruning thresholds (message count and token budget), post-pruning target token limits, and RAG parameters (episodic and contextual top-k, contextual keyword gates). Capability gating flags control which tool domains are exposed to each agent, providing an additional safety boundary.

9 Demonstration Workflows

Demonstrated workflows include instrument setup, single- and two-view centering, transmission correction, and tomography preparation. In centering workflows, images are acquired, segmented, and converted into stage-space corrections using geometry services based on acquisition sidecar metadata. Two-view centering uses orthogonal projections to ensure alignment in three-dimensional stage coordinates. Tomography workflows extend this loop to magnification selection and projection acquisition, producing datasets suitable for reconstruction and downstream analysis.

10 Performance and Benchmarking

Benchmark studies across multiple model configurations isolate the effect of planning quality on cost, reliability, runtime, and token consumption while holding architecture constant. Results indicate that planning fidelity dominates system-level efficiency; weaker models inflate tokens and runtime due to retries and replanning, while stronger models converge more quickly despite higher per-call costs. MCP-constrained execution ensures that failures remain safe and recoverable.

11 Threats to Validity

Benchmarks were conducted on a single representative workflow and may not capture behavior under substantially longer horizons or adversarial inputs. Performance depends on orchestration and prompt design choices, and cost and latency are subject to external API variability. Nevertheless, the qualitative trends observed—particularly the dominance of planning quality over actuation fidelity—are expected to generalize across similar autonomous microscopy deployments.

Algorithm 1 LangGraph-based multi-agent control loop (simplified)

```
1: Input: user request $u$, configuration $\mathcal{C}$
2: Initialize workspace $\mathcal{W}$; initialize state $S \leftarrow \textsc{CreateInitialState}(\mathcal{W}, u, \mathcal{C})$
3: $S[\texttt{next}] \leftarrow \texttt{supervisor}$
4: while $S[\texttt{next}] \neq \texttt{FINISH}$ do
5:     if $S[\texttt{next}] = \texttt{supervisor}$ then
6:         $S \leftarrow \textsc{PruneIfNeeded}(S, \mathcal{C})$
7:         $ctx \leftarrow \textsc{InjectMemory}(S, \texttt{supervisor}, \mathcal{C})$        ▷ RAG + lessons + world model
8:         $plan \leftarrow \text{LLM\_}\textsc{Plan}(u, S, ctx)$        ▷ JSON: tasks + assignments + routing
9:         $S \leftarrow \textsc{ApplyPlan}(S, plan)$
10:        $S[\texttt{next}] \leftarrow plan.\texttt{next}$
11:    else if $S[\texttt{next}] \in \{\texttt{instrument}, \texttt{image}\}$ then
12:        $agent \leftarrow S[\texttt{next}]$
13:        $S \leftarrow \textsc{PruneIfNeeded}(S, \mathcal{C})$
14:        $ctx \leftarrow \textsc{InjectMemory}(S, agent, \mathcal{C})$
15:        $taskBatch \leftarrow S[\texttt{assigned\_tasks\_for\_agent}]$
16:        $toolCalls \leftarrow \text{LLM\_}\textsc{Act}(agent, taskBatch, S, ctx)$
17:        $toolMsgs \leftarrow \textsc{ExecuteToolsViaMCP}(toolCalls, \mathcal{W})$
18:        $S[\texttt{last\_tool\_span}] \leftarrow \textsc{CaptureSpan}(toolCalls, toolMsgs)$
19:        $S[\texttt{next}] \leftarrow \texttt{world\_update}$
20:    else if $S[\texttt{next}] = \texttt{world\_update}$ then
21:        $patch \leftarrow \text{LLM\_}\textsc{ReduceToolSpan}(S[\texttt{last\_tool\_span}], S)$
22:        $S \leftarrow \textsc{DeepMerge}(S,\ patch.\texttt{world\_model\_patch})$
23:        $S \leftarrow \textsc{IndexArtifacts}(S,\ patch.\texttt{artifacts})$
24:        $S \leftarrow \textsc{UpdateTaskTrace}(S,\ patch.\texttt{task\_trace\_patch})$
25:        $S \leftarrow \textsc{AppendLesson}(S,\ patch.\texttt{lesson})$
26:        $S[\texttt{next}] \leftarrow \texttt{verifier}$
27:    else if $S[\texttt{next}] = \texttt{verifier}$ then
28:        $decision \leftarrow \text{LLM\_}\textsc{Verify}(S)$        ▷ JSON: done/fail/retry + routing
29:        $S \leftarrow \textsc{ApplyVerifierDecision}(S, decision)$
30:        if $decision.\texttt{status} = \texttt{failed}$ then
31:            $S[\texttt{retry\_count}] \leftarrow S[\texttt{retry\_count}] + 1$
32:            if $S[\texttt{retry\_count}] > S[\texttt{max\_retries}]$ then
33:                $S[\texttt{next}] \leftarrow \texttt{supervisor}$        ▷ escalate / replan / abort
34:            else
35:                $S[\texttt{memory\_context}] \leftarrow decision.\texttt{retry\_hint}$
36:                $S[\texttt{next}] \leftarrow decision.\texttt{retry\_route}$        ▷ instrument or image
37:            end if
38:        else
39:            $S[\texttt{next}] \leftarrow decision.\texttt{next}$        ▷ batch continue or supervisor
40:        end if
41:    end if
42: end while
43: Finalize workspace $\mathcal{W}$; export metrics, artifacts, and logs
44: Return: final state $S$
```