rvmc — A Research Workbench for Born-Digital Art

The Research VM Controller: browser-based, AI-assisted analysis and conservation of legacy software artworks

Marc Schütze
ZKM | Center for Art and Media Karlsruhe
July 2026

> **Draft — work in progress (July 2026).** I refresh this as the system grows. The core architecture and the decisions behind it are settled and a working prototype runs today, but the edges — some components, the wording, parts of the structure — are still moving; section 10 is my honest account of what is built and what is not. I share it early to invite collaboration, not as a finished product.

Abstract

A large part of the art of the last thirty years is software, and the machines it was written for — Windows XP, Mac OS 9, the vintage browsers and plugin runtimes — are disappearing. I have spent the last years restoring such works for exhibition at ZKM, and the lesson I keep relearning is this: **preserving a born-digital work means not archiving its files but keeping a living system able to run it.** Before I can decide how to keep a work alive, I have to understand what it actually depends on, and how it fails when a piece of that is missing — and the only reliable way to learn that is to run the work in a faithful machine and watch it closely.

This paper describes the **Research VM Controller (rvmc)**, the browser-based, AI-assisted workbench I built for exactly that task. A researcher creates a *project*, clones a period operating system from a sealed master image, boots it in an isolated environment, drives it through a screen in the browser, watches its network behaviour, writes small analysis tools in a web-based editor, and keeps the whole exploration versioned. Before that hands-on work, an automated **analysis** — the *artwork autopsy* — tears the uploaded work apart to reveal what it is, what it needs, and every network host it reaches; and an AI **analyst** can then drive the revival itself, booting the era-correct machine, running the work, and recording what happened — always on request, never resetting state on its own. rvmc grows out of ZKM's open-source *exhibition-vm-controller* (MIT-licensed), from which I deliberately removed one thing above all: the automatic roll-back, so that a research machine never silently undoes the researcher's — or the agent's — work.

1. The conservation problem

A large and growing part of contemporary art exists only as software. Net art, CD-ROM works, Flash and Shockwave pieces, virtual worlds, early Java–QuickTime experiments, browser-based installations — all authored against particular operating systems, plugins and network services, many of them now obsolete or gone. The *files* survive in the archive; the *conditions under which they become art* — the right OS, the right plugin version, the right server answering across the network — do not.

So conserving these works is, first, an act of **understanding before deciding**. Before I choose a strategy — migrate, emulate, re-create, document — I have to know what the work truly depends on: which libraries it loads, which addresses it contacts, which codecs it assumes, how it breaks when something is missing. I have learned not to trust the file listing for this. The dependency that kills

a work is usually the one that is *not* visible in its files — a server it quietly calls, a URL it assembles at run time. The only way I have found to surface those is to run the work in a faithful environment and watch it, and to be able to try things, break them, snapshot, roll back, and try again without ever endangering the original.

This is the opposite of **exhibiting** a work. An exhibition system keeps one known-good state running unattended for months and recovers from any crash on its own. A research system must do the reverse: expose state, preserve every intermediate condition I create, log what it does, and never undo my work behind my back. A *virtual machine* (VM) — a whole computer, operating system and all, running as software on a host — is the natural vehicle: it lets an original environment live on long after its hardware and software stack are gone, and it can be copied, frozen, inspected and shared. rvmc is a workbench built around that.

---

2. Background: from exhibition to research

I did not build rvmc from nothing. It grows directly out of a system I had already built at ZKM — the *exhibition-vm-controller* (evmctl), a framework for running historical digital artworks unattended in an exhibition, released as open source under the MIT licence. That system runs a work inside a VM, captures a known-good *snapshot* (a saved copy of the machine's exact state), checks with a periodic "heartbeat" that the work is still alive, and **rolls back to the snapshot automatically** the moment anything goes wrong. It boots straight into the artwork, full-screen, with no operator present — exactly right for a gallery, and exactly wrong for a researcher.

The workbench keeps that system's foundations: the same virtualisation technology, the same snapshot mechanism, a host-side controller written in Python, and a small helper program inside each guest machine (the *guest* — the operating system running inside a VM). I factored the shared parts into a common core, `vmctl-core`, that the research system builds on today, and I plan to bring the exhibition system onto the same core (section 10). But I deliberately removed three things from the exhibition side:

- **The autonomous full-screen player.** The interface is now a workbench in a web browser.
- **Automatic roll-back.** Nothing resets a VM on its own; state is sacred.
- **Physical-installation autonomy** — the hardware sensors and watchdog timers a gallery needs.

One rule follows from all three, and everything else in the design follows from it: *a research VM must never silently reset state while I am working in it.*

---

3. Design principles

These are the rules I hold the system to; where I have had to choose, I chose in their favour.

1. **Researcher-controlled state.** Snapshots, roll-backs and resets are *always* explicit actions. Automatic snapshots accumulate over time; they never overwrite the researcher's work.
2. **Isolation by construction.** Each VM runs on its own private, isolated network. The control link between the workbench and the inside of a VM — the channel the workbench uses to drive the guest — is *host-only*: it is not reachable from any network the artwork itself can see.
3. **Reproducibility.** Master images (the sealed, read-only OS templates) are configured with period-appropriate emulated hardware, so an image behaves consistently on any host and into the future — a precondition for citable conservation work.
4. **Configuration-driven and portable.** One master image, one configuration file. Nothing assumes a particular institution's infrastructure, so any institute can deploy it on its own equipment.
5. **One tool library, many surfaces.** The same collection of analysis tools is offered to the AI chat assistant, to an autonomous **analyst agent** that can carry out a whole task on request, and to the researcher working in the code editor. A tool is written once, defined in one place, and available everywhere.

6. **Legal, attributable sharing.** Tools and images are shared between cooperating institutes as self-describing bundles that carry their own licence and record of origin.

---

4. System architecture

I separate a shared **control plane** (run once per installation) from per-project **work areas** (one for each research project or student group).

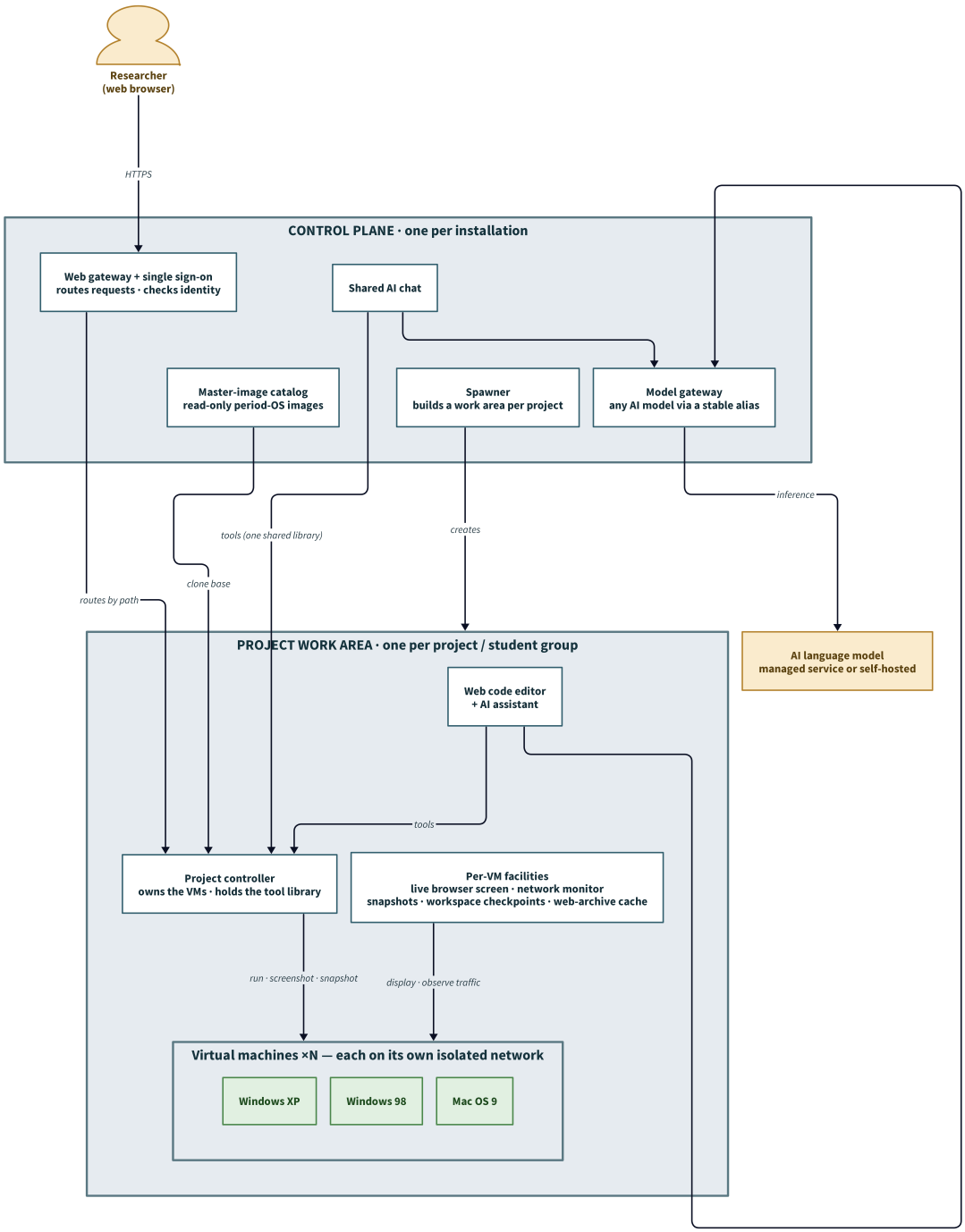

Figure 1: **Figure 1.** The two tiers of rvmc. A shared control plane creates and fronts an isolated work area for each project; every work area runs one or more period-operating-system virtual machines, with a single tool library feeding both the AI chat and the code editor.

Control plane (one per installation). The control plane bundles the services shared across all projects: a registry of projects; a catalog of read-only master images; an idempotent *provisioner* that

creates, reconciles, and tears down a fresh, self-contained work area for each project (wiring up its identity groups, AI tool-server registration, and optional web-archive cache in one repeatable step) — exposed both through the admin web surface and as admin-only tools an operator or assistant can call; a single sign-on layer that checks who a user is before letting any request through (provided here by *Authentik*, an open-source identity manager); a web gateway that routes incoming requests and handles encryption (the open-source reverse proxy *Traefik*); a single shared AI chat interface; and a *model gateway* (the open-source tool *LiteLLM*) that lets any part of the system reach a chosen AI language model through a stable, swappable name.

Project work area (one per project). Each work area contains: a controller (a small web service, written in Python with the *FastAPI* framework) that owns that project's VMs; a live screen for each VM shown in the browser; a browser-based code editor; a network monitor; snapshot and checkpoint management; and a local cache of archived web content for that project.

The data model. This is where I broke most sharply from the exhibition system, which managed exactly one VM. It has three nested layers:

- A **master image** — a sealed, read-only template of a period operating system (which OS, what emulated hardware, whether an in-guest helper is present).
- A **project** — a workspace built on a thin "copy-on-write" private copy of a master image (instant, almost no disk used). A project holds its owner/group, a version-controlled folder of tools, notes and data, shared data folders, and its collection of tools.
- One or more **VMs** — running copies of the project's image, each with its own browser screen, its own isolated network and monitor, and its own snapshots.

A "copy-on-write" copy means the new VM starts as an empty layer stacked on top of the shared, read-only master image: it reads through to the master for anything it has not changed, and only the *changes* take up space. Making such a copy is therefore near-instant and nearly free in disk terms, and a project can run several at once — Windows XP and Windows 98 side by side, say.

The artwork's lifecycle. In practice a project *is* one artwork, and it moves through stages recorded on the project's own durable folder (its *drive*): **analysis** — the automated autopsy that studies the uploaded work (section 6); **workbench** — the hands-on revival described in this paper; and, as the eventual destination, **exhibition** — handing a verified, frozen result to the exhibition controller. The stage is a marker, not a gate: stepping back from workbench to analysis is normal.

Addressing and access. Everything lives under a single web address, organised by path (`…/admin`, `…/project_<id>/code/`, `…/project_<id>/screen/<vm>/`, and so on), so that adding a project needs no new web addresses or security certificates. The gateway routes each path to the right component, and the sign-on layer in front limits each user to their own project(s).

---

5. Core mechanisms

Making VMs — instant private copies. Master images are sealed and read-only. A new VM is the "copy-on-write" layer described above, stored in QEMU's standard disk format (`qcow2`). (QEMU is the open-source software that emulates a whole computer; on Linux it is accelerated by the kernel's built-in virtualisation, *KVM*.) Creating a VM is instant and costs almost no disk; resetting one is a single command that discards the layer and starts fresh. I designed two ways of hosting the VMs behind one configuration setting: running them on a host or dedicated server through libvirt, or running them directly inside the project's container. The libvirt-based path is implemented today; the in-container option is on the roadmap (see sections 9 and 10).

Checkpoints — two independent kinds. VMs take *snapshots* (saved copies of the machine's full state, including memory) on command — created, listed, reverted, and deleted explicitly. An automatic, timed snapshot loop that keeps the most recent few and discards the oldest is designed and on the roadmap (section 10). Separately, a *saved checkpoint* will record the researcher's workspace —

their tools, notes and data references — using ordinary version control (the same *git* system used in software development), noting alongside it which VM snapshot was current at the time; this is designed but not yet implemented. The two are kept independent on purpose: restoring a workspace checkpoint brings back *files*; rolling a *VM* back to a snapshot is always a separate, deliberate act. There is no surprise reset of a live machine.

The screen in the browser. Each VM's display is shown live in the web browser using *noVNC* — a browser implementation of VNC, the same screen-sharing technology used for ordinary remote desktops — bridged to the VM by a small connector. Because this captures the screen at the host level, it needs no cooperation from the (often very old) software inside the VM. Quirks of legacy systems are handled explicitly, for example scaling the picture in the browser rather than asking a 2001-era operating system to change its resolution, and giving it a precise mouse so the cursor lines up with the browser window.

Getting data and commands in and out. Files move in and out by several routes: direct upload (written into the guest by the in-guest helper); a **CD-ROM built on demand from a host folder** — a standard ISO9660 disc (with Joliet and Rock Ridge extensions) that every guest operating system understands, rebuilt and re-attached when the folder changes — which keeps a single mechanism working uniformly from Windows XP to Mac OS 9; or launching the VM's own web browser at a given address while the network monitor records what it fetches. (Writable shares that appear as a live network drive — SMB for legacy Windows, virtiofs/9p for Linux — are on the roadmap; today the on-demand CD plus direct upload cover the read-in path.) Just as importantly, a researcher can export their workspace, notes and captured data at any time, in as many independent copies as they like — nothing locks results inside the system. Where the guest can support it, a small helper program runs inside the VM and talks to the workbench over a **host-only private channel** — a direct line between host and guest that, by design, sits on no network the artwork could ever reach. I built the channel *multi-lane* — a small pool of independent lanes, so that a long-running operation on one lane never blocks control or screen traffic on another (control and file-transfer lanes pinned, heavier calls spread across a worker pool); the helper and host exchange newline-delimited JSON-RPC messages. In practice I run it at a **single lane** today, and not because I want to: more than two of the underlying virtual serial ports destabilise the driver on modern Windows (10/11), and I have not yet found the fix. It is one of the open problems I am least happy about (section 10). (I have specified and prototyped a more compact length-prefixed protocol, *RVMC-WIRE/1*, as a future upgrade.) On Windows this helper can run commands, read and write files, read the system registry, photograph the screen, and — most usefully — *install software from a virtual CD by clicking through its installer dialogs automatically* (using a Windows automation language, AutoIt, for the clicking). A counterpart agent now also runs inside **Mac OS 9** — written in the era's MacPython (Python 2.3) — and can take screenshots, send keys and mouse events, read and write files, list and control running applications (through AppleEvents), run AppleScript, execute code, and surface the guest's logs; the workbench advertises each guest's in-guest scripting floor so an assistant knows, at discovery time, what language and version it is working against.

Watching the network. Each project's VMs sit on their own isolated network with the host as the gateway, which gives three independent, switchable layers of observation:

- **Passive recording** of all network traffic to a standard capture file, for later analysis.
- **Web-traffic interception and replay.** A per-project caching proxy — my *wayback-cache-proxy* — sits in the path of the VM's web requests; I point the guest's browser at it explicitly. To read traffic that is encrypted (HTTPS), the workbench installs its own trusted security certificate inside the guest, so even secure connections — to servers dead for fifteen years — can be intercepted, cached, and *replayed* from web archives (via the open-source *pywb*), with optional period-accurate speed throttling. Answering the question that comes before replay — *what did that dead server actually serve, and when?* — is a separate, shared piece I built for it: **wayback-mcp**, a small MCP server

over the Internet Archive's Wayback Machine that the analyst queries to find the right archived snapshots, which the proxy then serves back into the running VM (Appendix C).
- **Stand-in services** *(planned)*. When an artwork's original servers are gone entirely, the workbench will be able to stand in for them, so the work still "sees" a live internet answering its requests. During the source exhibition these were written ad hoc per artwork; folding them into the workbench as a reusable layer is on the roadmap.

---

6. The analysis and the agents

Two capabilities turn the workbench from a VM console into a research instrument: an automated *analysis* that studies a work before anyone touches it, and an AI surface — a shared tool library and an agent — that acts on what it finds.

Automated analysis — the autopsy. Before a researcher (or the AI) touches a VM, an automated *analysis* step studies the uploaded work and writes a plain conservation report. It runs as its own service (the companion **artwork-autopsy** project) over the same per-project drive: it unpacks the bundle, identifies its file formats and entry point, reads its dependencies, and — the part I care most about for a work that no longer runs — surfaces **every network address it tries to reach**. Naming a dead address is only half the answer, though; the other half is *what it served* — and the analyst settles that during the analysis itself, by querying the web archive. Through the shared **wayback-mcp** it asks the Internet Archive what an address returned around the work's own era and fetches the actual archived bytes as ground truth, so the report records not merely that a server is dead but what it answered and how to serve it back. I treat this as integral to the autopsy, not a downstream nicety: it is the difference between a report that says "this endpoint is gone" and one that hands the workbench the concrete thing to replay.

Where the work is more than plain files the analysis goes deeper: it decompiles Director/Shockwave, Flash, Java, .NET and Unity, reads native executables, and can carve an artwork's own data out of a self-contained projector. The coverage is not only legacy: the same machinery reaches modern web-based art — an Electron application (a Chromium and Node runtime the analysis unpacks and pins to its bundled version) or a browser extension across its format eras — where the preservation problem has exactly the same shape, a runtime frozen at one moment and servers that may already be gone.

Deterministic tools *extract* the raw signals; an AI *analyst* then reads that evidence and reconstructs what the work is, its crucial parts, and the network endpoints — judging, for example, whether a web address is one the work assembles at run time or an inert default. The report suggests the era-correct base machine and the runtimes to stock; because an installer can itself be an artwork, it pauses to ask the researcher before treating a bundled runtime as the work rather than the piece. Its output — a report, a revival runbook, and suggestions — lands on the drive and feeds the workbench: it is what the analyst agent and the researcher act on.

From the same evidence it also composes three audience-specific views: a **dependency manifest** for the conservator (what must be provisioned), an **asset catalogue** for the researcher (what the work is made of), and — because this is a tool for art historians as much as engineers — **the artist's hand**, the artist's own words harvested from the source and the decompiled code: their comments, their authorship and copyright lines, the contact URLs, the voice inside the work. That last view is deterministic and interpretive in equal measure — the pass *extracts* the words, the historian reads them.

The AI and tooling surface. Each project offers a single collection of tools — controlling the VM (take a screenshot, send a keystroke, take a snapshot, insert a CD), acting inside the guest (the helper program's abilities above), and the project's own custom analysis tools. Standard open-source forensic and analysis tools can be packaged the same way and offered alongside them. This collection is exposed through the *Model Context Protocol* (MCP), an emerging open standard for

giving AI assistants access to tools, and the *same* collection feeds the chat assistant, the AI built into the code editor, and — for power users — a researcher's own external agent. The actual AI model is reached through the model gateway by a stable alias rather than by name, so the underlying model can be swapped in one place.

Beyond offering these tools for a person or a chat model to call one at a time, I gave rvmc its own **analyst agent**. Given a goal in plain language — *"revive this work and tell me whether it runs"* — it carries out the whole sequence itself, using the same tool collection: it reads the autopsy report, compiles it into an ordered plan, boots the suggested era machine, launches the work, watches the network against the endpoints the analysis predicted, judges whether the work is actually alive, and records the outcome back onto the drive. Crucially, this is *task* autonomy, not *state* autonomy: the agent drives the workbench on request, but — like everything else here — it never snapshots over or resets a running machine behind the researcher's back. The agent is itself offered as just another tool, so the chat assistant (or a researcher's own external agent) can hand it a whole job rather than steering each step by hand.

Access follows the user's identity across three tiers: a *researcher* works within their own project's research tools; a project *admin* (a teacher) additionally holds the destructive and onboarding operations for that project; and a platform *master* operates across every project. A researcher can do the day-to-day work but not hard-reset, delete, or onboard others.

A programmable interface. For researchers who want to run their own software, or who want to drive the workbench from scripts, the controller also offers a conventional web API (a *REST* interface, documented in the standard *OpenAPI* format) protected by a per-project access token, exposing only the project-scoped operations.

---

7. Legacy operating-system coverage

I group the target operating systems by how far I have got with each.

Status	Operating systems
Works today — built, agent-verified	Windows 98 SE · Windows 2000 · Windows XP / Server 2003 · Windows 7 · Windows 8.1 · Windows 10 (22H2) · Windows 11 — each with a working in-guest agent · Mac OS 9.2.2 (PowerPC), with an in-guest MacPython agent · Mac OS X 10.4 Tiger (PowerPC) · Mac OS X 10.6 Snow Leopard (Intel)
In progress — stabilising	Windows 98 SE period networking (the agent runs; the era NIC is being finalised)
Maybe later — researched, on the roadmap	Windows ME · classic Mac OS 8 / 7.6 and 68k Macintosh (OldWorld emulation)

Modern Linux does not need a sealed base image — it runs directly on the host — so the golden bases concentrate on the legacy Windows and Macintosh systems that genuinely need period emulation. (Per-OS status moves quickly; the build evidence kept in the repository is the authoritative record.)

The constant principle is **period-appropriate hardware**: each old operating system is given the virtual devices its own drivers were written for, rather than modern ones, so the images stay portable and behave consistently over time.

---

8. Shareable tooling and inter-institutional federation

I keep two related units distinct. A **plugin** is the unit that captures the result of a conservation effort and moves it from research toward exhibition: a small, versioned directory with a manifest and a

single entry point, loaded through a shared plugin spec that both the research and (in future) the exhibition controller understand. A plugin can *expose* one or more **MCP tools** — the host- and guest-side actions an assistant calls — and may reference catalogued **runtime tools**. The plugin is the deliverable; the tools are its building blocks.

Those runtime tools are the period binaries and installers (Flash and Shockwave, the Java runtimes, QuickTime, Silverlight, VRML players, .NET) the workbench keeps as a curated registry: each pinned to a version, checksummed, and tagged with its licence, end-of-life status and the quirks of actually running it — post-2021 Flash, for one, refuses to render until its own dead check-in servers are sinkholed. And the registry is queryable, not just a shelf: the runtimes are linked into a dependency graph — *depends-on*, *bundled-with*, *an-alternative-to*, *reads-this-format* — so the analyst can resolve a work's stated need ("Flash 8 on Windows XP") to a ranked, runnable stack: the exact binary, the chain it depends on, and its conservation quirks. That is the work's *technical environment*, in the preservation sense, made resolvable.

A scaffolding command (`rvmc plugin new <name>`) drops a ready-made starting skeleton — manifest, entry point, and a small web view — into the project's workspace on demand. I wrote a real worked example that ships as its own plugin repo in the vmctl `plugins/` subgroup: a VRML-conservation plugin that encodes the bring-up recipe for a 1990s VRML world in Internet Explorer 6 (clone the base, enable the web-archive cache, install the period 3D browser plugin, launch, snapshot) — a plugin *describes* the recipe and the controller executes it.

Packaging a finished plugin into a self-describing **bundle** (description, code, version, licence) and a **registry** that records each bundle's origin and licence are designed but not yet built (section 10); today plugins carry their own manifest and licence metadata in the repository. By policy, sharing is **inter-institutional and academic** — and the same applies to the master OS images, which may contain licensed operating systems and copyrighted artwork (see section 9).

---

9. What it takes to run rvmc at an institute

rvmc is built so that the institutions actually doing the conservation work can host it themselves. Deployment follows a phased path from prototype to independent operation.

Hosting phases.

- **Prototype phase (now).** The system currently runs on infrastructure I have to hand, and can stay there while the design and tooling stabilise and while a small number of cooperating groups try it out. Those groups can start straight away, with nothing to install on their own side.
- **Institutional phase (target).** For real teaching and research use, each participating institute should host its own installation. The system is deliberately portable for exactly this reason: nothing in it assumes the prototype's infrastructure. Moving an installation is a matter of supplying a configuration file and connecting the institute's own login system — not rewriting anything.

Beyond hosting, there are three resources each institute needs to plan for.

1 — A server sized for the number of simultaneous users. The dominating cost is the running VMs, not the rest of the system. Because each student's or researcher's VM is a thin copy-on-write layer over a shared master image (section 5), *disk space is nearly free*; the real budget is **memory and processing power for the VMs that are switched on at the same time**. Individual legacy machines are modest (a Windows XP VM is comfortable in 1–2 GB of memory), so a single well-equipped server can carry a whole class or research group. Two practical points shape the deployment:

- **Hardware virtualisation must be available** on the server (the standard Linux KVM feature). The system is designed so that, if the institute's cloud or container platform does not provide it, the VMs can instead run on a dedicated physical server — the choice being one configuration setting rather than a redesign. The dedicated-server (libvirt) path is implemented today; the in-container path is on the roadmap (section 10).

- **Size for the peak number of VMs running at once,** not the total number of projects. Switched-off VMs cost only disk space, which the automatic-snapshot housekeeping keeps bounded.

2 — A shared software stack, with its legal implications understood. rvmc's value compounds when institutes share master images and analysis tools — but those carry obligations the software cannot waive on the user's behalf:

- **Operating systems.** Master OS images contain proprietary operating systems (Windows, Mac OS). Each institute is responsible for holding valid licences for what it runs and shares. The system records where an image came from; it does not grant any rights to it.
- **The artworks themselves** are under copyright held by artists or collections. Running and studying them — and exchanging the resulting VM images — must rest on the appropriate agreements with the rights-holders, agreements that institutes will typically need to work out together with their own legal offices.
- **Why sharing is deliberately gated.** For these reasons, sharing of tools and images is **inter-institutional and academic by design**, and every shared bundle carries its licence and origin so an institute can see, before importing anything, what terms it is taking on. The workbench's *own* code builds on an MIT-licensed base and is intended for open release; the *images and artworks* moved through it are not automatically free to redistribute.

3 — Access to AI computing power. The AI features (the chat assistant and the assistant inside the code editor) need an AI language model to talk to, reached through the model gateway by a stable alias. An institute has three options, in increasing order of self-reliance:

- **Managed academic AI service** — for example a national or European research-computing service the institute already has access to. This is the lowest-maintenance option, and the one I recommend.
- **A shared connection to a commercial AI provider**, behind the same alias.
- **Self-hosted models** on the institute's own graphics hardware, where data sensitivity or cost make that preferable.

Because every part of the system addresses the model by alias rather than by name, an institute can start on a managed service and move to self-hosting later by changing a single entry — with no change to the workbench itself.

Whichever option an institute chooses, the model gateway can attach a **separate AI budget to each group** — a class, a cohort, a research project — so cost is metered and capped per group rather than pooled. In a shared deployment across institutions this is how one hosting institute can provide the infrastructure while each participating group brings, and pays for, its own AI access: the group's identity carries its own gateway key, and its spending stays its own. Running the old software is cheap; the *analysis* is where the cost sits, so putting it behind a per-group budget is what keeps it affordable and controllable.

---

10. Implementation status

This is my honest ledger. As of July 2026 the following are **built and proof-tested**: the instant copy-on-write VM mechanism over the libvirt hosting path; manual snapshots (create, list, revert, delete); the live in-browser screen; the multi-lane in-guest helper and its host-only channel on Windows, and an in-guest MacPython agent over emulated networking on Mac OS 9; the on-demand CD-ROM and direct-upload data routes; per-project network capture and the per-project web-archive cache (caching, HTTPS interception via an in-guest certificate, and replay); the shared tool collection feeding both the chat assistant and the code editor; the per-project provisioning, three-tier access control, and project-scoped API tokens; the programmable web API; a worked example conservation plugin; and verified master images across a broad range of legacy Windows — Windows 98 SE, 2000, XP / Server 2003, 7, 8.1, 10 and 11, each with its in-guest agent — plus Mac OS 9.2.2 and Mac OS X 10.4 Tiger on emulated PowerPC and Mac OS X 10.6 Snow Leopard on Intel (build evidence kept

in the project repository; modern Linux needs no base image, running directly on the host). The shared control plane — the gateway, the single sign-on, the model gateway, the chat interface and the per-project provisioner — runs on the prototype installation, which is currently wired to my own infrastructure. Also built and exercised: the automated **analysis** (the *artwork-autopsy* service — unpacking, format and dependency analysis, the network-reach survey, deep decompilation of Director, Flash, Java, .NET, Unity and native binaries, and the AI analyst that reconstructs the work), which writes its report, revival runbook and suggestions onto the project drive. The analysis is the mature part: on a benchmark set of eleven works spanning six runtime families — Java with QuickTime, Flash, Director/Shockwave, classic-Mac, native Win32, and web — it identified the entry point and the runtime correctly for ten, on an EU-hosted, sovereign model stack, with no fabricated findings.

One honest exception to "exercised." rvmc's own **analyst agent** — which turns that report into an ordered plan and drives a revival end-to-end: booting the era machine, launching the work, checking it against the network endpoints the analysis predicted, and recording the outcome back onto the drive — is *wired*, and each of its parts (the VM control, the in-guest agent, the archive proxy, the model loop) is proven individually. But I have not yet run the full autonomous revival against a real work. I am confident it will work, because every piece it depends on already does; I have simply not yet earned the right to say that it does. That first end-to-end revival on a real artwork is the next milestone, and I would rather report it honestly as pending than claim it early.

What I have not built yet. These are designed but not yet built: automatic timed snapshots with retention housekeeping; git-backed workspace checkpoints; the in-container (direct-QEMU) hosting backend; writable live network shares (SMB for legacy Windows, virtiofs/9p for Linux); reusable stand-in services for absent servers; packaging plugins into self-describing bundles plus a registry for inter-institutional exchange; the shared file-server that exchange depends on; Classic Macintosh systems (Mac OS 8 and 68k); an in-guest helper for modern 64-bit Windows; current-standard secure-boot master images; brokering single sign-on to institutional federation (eduGAIN and similar); widening the in-guest channel beyond a single lane on modern Windows; the automatic hand-off of a verified result to the exhibition controller; and the path that brings the original exhibition system onto the same shared core.

---

11. Where this sits: related work

I did not invent most of the pieces here, and it matters — to me, and to anyone reading this as a research claim — to be precise about which part is actually new.

Emulation as a service is the closest infrastructure. The EaaS / EaaSI line (Klaus Rechert and colleagues, Freiburg and Yale; the Software Preservation Network) runs legacy environments in the browser and shares them across institutions, and its Universal Virtual Interactor automatically *matches* a digital object to a pre-built environment — for a *known* file format, from a hand-curated library, when there is enough documentation to identify it. That is real automation, of environment *selection*[1]; as of its 2025 roadmap it has added no AI reasoning, no unknown-bundle triage, no runbook authoring — the environments are still configured by hand. Olive and VMNetX (Mahadev Satyanarayanan, CMU)[2] stream access to human-curated VMs and discover nothing. The virtual machine as the vehicle of preservation is well-trodden ground; I am standing on it, not clearing it.

The nearest neighbour is recent and close, and I want to name it plainly. Rechert and Gieschke (*International Journal of Digital Curation*, 2024) put a language model in front of an emulated machine: it reads the screen, writes a step-by-step plan — each step a command with its expected outcome —

[1] Euan Cochrane et al., "Towards a Universal Virtual Interactor (UVI) for Digital Objects," in "iPRES 2019: 16th International Conference on Digital Preservation," special issue, *iPRES 2019: 16th International Conference on Digital Preservation*, 2019, 191–200, https://ipres2019.org/static/pdf/iPres2019_paper_128.pdf.

[2] Mahadev Satyanarayanan et al., *One-Click Time Travel*, technical report CMU-CS-15-115 (2015), https://olivearchive.org/static/documents/CMU-CS-15-115.pdf.

drives the VM to execute it, and uses a second model to check whether each step worked, replanning on failure. That is strikingly close to the workbench agent here, which compiles a report into an ordered plan and drives the revival[3]. So I do not claim that using a model to author and drive a revival plan on an emulator is new: it is published, and by the very group whose emulation work I build beside — the same group has separately shown language models distilling how to *operate* a known system from recorded sessions[4].

Web replay and dead-server revival is where Rhizome has done the closest work to the exhibition side — Webrecorder / Conifer and oldweb.today (Dragan Espenschied, Ilya Kreymer) archive the live web and revive the servers a net-artwork still calls; their 2018 Conifer prototype[5] chained archiving, server-preservation and container revival, for a *known* stack. It is the strongest prior art for the workbench's archive proxy.

And the conservation field has named my problem — and does it by hand. Tate's software-based-art group (Patricia Falcão[6], Tom Ensom[7]) describes the exact task of the autopsy — the derivation of a recipe for reconstructing the technical environment a work needs to run — and calls it *laborious and highly specialised* manual labour; the wider literature — the Variable Media questionnaire[8] among it — still treats the automated characterisation of an unknown work as an open problem. The format tools I lean on — PRONOM/DROID, JHOVE, Siegfried — answer *what a file is*, never *how to run it*; the re-running engines a revival needs — Ruffle, ScummVM, DOSBox, ProjectorRays, CheerpJ, X_ITE — supply the engine but say nothing about which one a given work wants.

And the field has *tried* to automate the environment side before — which is the cautionary tale I build against. The KEEP project (Keeping Emulation Environments Portable, EU, concluded ~2012) and its TOTEM registry set out to document the full technical-environment dependency chain — hardware, operating system, plug-ins, libraries — so an emulation framework could compute the environment a work needs[9]. That is the closest prior attempt at what the autopsy does, and its fate is instructive: capturing every dependency of every dependency by hand is a combinatorial task that never reached critical mass, and the registry stalled when the funding did. The lesson I take is not that the goal is wrong but that the *depth* is. My own tool learned the same thing the hard way: pointed at a Java application it first reported the entire bundled dependency tree — dozens of third-party library jars — as things a conservator must supply, and pointed at a Wine-wrapped work it listed the wrapper's whole internal runtime as requirements. Both are the TOTEM failure in miniature, and the fix was to *subtract* rather than enumerate — a bundled runtime is one thing to provide, not a graph to catalogue. The current pan-European effort, EMBARK (2025–2029), names the same preservation crisis but is a networking action — conferences, training schools, open calls — with, as yet, no shared executable method[10]. The significant-properties literature has meanwhile turned against exhaustive

[3] Klaus Rechert and Rafael Gieschke, "Preserving Secondary Knowledge — Using Language Models for Software Preservation," *International Journal of Digital Curation* 18, no. 1 (2024), https://doi.org/10.2218/ijdc.v18i1.930.

[4] Klaus Rechert et al., "Preserving Users' Knowledge of Contemporary and Legacy Computer Systems: Automated Analysis of Recorded User Activity," in "iPRES 2024: International Conference on Digital Preservation," special issue, *iPRES 2024: International Conference on Digital Preservation*, 2024, https://doi.org/10.21428/5676bf2d.ff6a2943.

[5] Ilya Kreymer, "A Prototype of Automated Web Archiving, Emulation and Server Preservation," August 28, 2018, https://blog.conifer.rhizome.org/2018/08/28/automation-emulation-server-preserve.html.

[6] Patrícia Falcão et al., "An Exploration of Significance, Dependency, And Virtualization in the Conservation of Software-Based Artworks," *The Electronic Media Review* 4 (2016), https://resources.culturalheritage.org/emg-review/volume-4-2015-2016/falcao/.

[7] Tom Ensom, "Revealing Hidden Processes: Instrumentation and Reverse Engineering in the Conservation of Software-Based Art," *The Electronic Media Review* 5 (2018), https://resources.culturalheritage.org/emg-review/volume-5-2017-2018/ensom/.

[8] A. Depocas et al., *Permanence Through Change: The Variable Media Approach* (Guggenheim Museum Publications, The Daniel Langlois Foundation for Art, Science,, Technology, 2003), https://www.variablemedia.net/pdf/Permanence.pdf.

[9] Janet Delve and David Anderson, "Preserving Games Environments via TOTEM, KEEP and Bletchley Park," in *Preserving Complex Digital Objects*, *Preserving Complex Digital Objects*, ed. Janet Delve and David Anderson (Facet Publishing, 2014), https://wiki.softwareheritage.org/wiki/TOTEM.

[10] EMBARK (EU COST Action), "EMBARK: Enabling a Community for Media Art Conservation and Knowledge," 2025, https://www.dpconline.org/news/dpc-joins-embark.

characterisation on principle: significance is fluid and contextual, and appraisal should decide what is essential and accept loss elsewhere[11], rather than trying to record everything.

So the honest claim is narrow, and I mean to keep it that way. What is new here is not AI in preservation, nor the VM as a base, nor authoring a plan to drive an emulator — each of those has prior art I have cited. It is two things. **First, the reasoning from an *unknown, undocumented* bundle to a proposed runnable environment** — taking a work nobody has catalogued and inferring its identity, its era-correct machine, its period runtimes, the hosts it reaches, and what a dead server once served. **Second, carrying that one result through a single system, from analysis to a research workbench to a locked exhibition.** The field has named this problem and, so far, solved it by hand. This is an attempt to automate the *understanding* — the part before the decision — and to keep a human in the loop exactly where judgment belongs.

---

12. Conclusion

I built rvmc because restoring born-digital art taught me that the hard part is never running old software — a VM is a VM — but *knowing what a work needs before deciding how to keep it alive*. The autopsy in this system exists to make that knowledge cheap and honest. When I fed it a Macromedia Director piece I had only as a Mac *StuffIt* archive, taken as found (Appendix E), the deterministic pass told me plainly what it could see — a classic Mac application, Director/Shockwave, Mac OS 9 — and just as plainly what it could *not*: the work reaches the network, but the address is assembled at run time. It did not guess. The analyst then decompiled the cast and read the destination out of the code — the work calls home to its own back-end on a university server, long dead. Nothing in the artwork's files announced that dependency; it lived inside compiled Lingo. And it is exactly the kind of dependency that decides whether the piece lives or dies: name the dead host and serve it from an archive before the first run, or watch the work fail silently on the gallery wall.

That is the whole argument in one work. In my earlier restorations for the *Choose Your Filter!* exhibition I settled on a two-part strategy for these external dependencies, and I built rvmc to support both. Where I can, I **eliminate the dependency** — mirror what the work needs into the machine, so it runs almost maintenance-free. Where I cannot, because the work needs live and unpredictable data, I **stand a replacement service outside the VM**, so that when the outside world changes, only that layer needs a fix and the artwork itself is never touched. The workbench's per-project archive proxy and its planned stand-in services are the general form of that lesson: a place to put the world a dead work still expects to find.

Two questions I have not resolved, and do not want to paper over. The first is **artistic integrity**. Every one of these methods — local mirrors, proxies, archive sources — changes the data environment a work runs in, even when its code is left untouched. Where I can, I make those decisions in dialogue with the artist; where I cannot, the least I owe is a justified, transparently documented choice, and the workbench is built to record exactly what was changed and why. The second is **reach**. The architecture here is proven, but it still asks for real technical knowledge to stand up, and most museums and archives do not have that to spare. So the work that matters next is as much about lowering the barrier — prefabricated machine templates, reusable monitoring, a clean hand-off from the research workbench to the locked exhibition runner — as it is about supporting one more operating system.

And I want to be plain about the limit of what I can claim from a handful of works: this system earns its validity only by running a *lot* of art through it. Artists are not software engineers, and that is exactly the difficulty — they solve problems in ways an engineer never would, reach for a runtime in a manner its authors never intended, lean on a side effect that happens to look right on screen. Every such work teaches the autopsy something its modules did not yet know. So the tool grows the way the problem demands: I build an **artwork-specific module** to handle what one work does, then

[11] Devin Becker, "Metaphors We Work by: Reframing Digital Objects, Significant Properties, And the Design of Digital Preservation Systems," *Archivaria* 85 (2018): 6–37, https://archivaria.ca/index.php/archivaria/article/view/13628.

generalise it once a second and a third show the same shape. I am in the middle of exactly that now. A work often arrives carrying its whole runtime beside itself — one I am analysing ships an entire Java runtime of some two and a half thousand files — and without care the analysis loses the artwork inside the software it came wrapped in. So the newest work is a pair of modules that tell the two apart: one recognises a bundled runtime from its structure, the other, an agent, judges the harder cases — and neither *removes* anything; the runtime is only set aside for the analysis, and still runs in the VM. It began as a Java problem and is already generalising to the libraries and vendored code that works from other ecosystems arrive wrapped in. The corpus is the test suite; the surprises in it are the specification. This is also why I share the system early and openly: I cannot validate it alone.

And running that much art is cheaper than I expected. A full analysis is millions of tokens, nearly all of them *input* — the work's decompiled code and evidence going in, a short structured report coming out — so on an academic model service the whole corpus costs next to nothing, and only a run against a frontier commercial model reaches tens of dollars. What paces me now is throughput, not money: the analysis could go faster than it does, but I hold it to about eight requests a second against the model service, so a large corpus is bounded by that ceiling, not by its price.

Underneath all of it is a single claim I have come to believe. **Preserving born-digital art is not archiving old files; it is operating living systems** — systems robust enough to survive the unforeseen and transparent enough for the next conservator to understand and change. A virtual machine, a network monitor, an honest autopsy, and an agent that drives none of it without being asked are, I think, a workable foundation for that: for the historical web meeting today's infrastructure, and for the new generative works that will need the same care sooner than we expect.

---

13. Availability, licensing and acknowledgements

Availability and licensing. The software is open source under the MIT licence of the exhibition-vm-controller it builds on, published on GitHub under the ZKM org — **github.com/zkmkarlsruhe** (`artwork-autopsy`, `research-vm-controller`, `exhibition-vm-controller`, `vmctl-core`, `wayback-mcp`); this documentation lives at **docs.vmctl.org**. The material that flows *through* the system — master operating-system images and artwork-bearing VMs — is not automatically redistributable and is shared only between cooperating institutes under appropriate agreements (see sections 8 and 9).

Lineage. rvmc extends ZKM's *exhibition-vm-controller* (MIT-licensed; archived at DOI 10.5281/zenodo.18652760), whose authorship and copyright notice are preserved as the licence requires.

Stewardship and collaboration. rvmc is a ZKM Open-Source project (MIT). I lead it as head researcher, and ZKM | Center for Art and Media Karlsruhe stewards it for long-term continuity. It is taking shape in close collaboration with researchers in the history and conservation of media art, whose needs and feedback shape its design directly — the workbench exists to be useful to exactly that work.

Acknowledgements. I thank Inge Hinterwaldner for detailed feedback on this text and for helping steer the tool toward real research practice.

Appendix A — Technical notes

For technically inclined readers, here are the concrete choices behind the plain-language descriptions above.

- **Virtualisation:** QEMU/KVM. Master images and private copies use the `qcow2` disk format with backing-file overlays (copy-on-write); a reset recreates the overlay.
- **Pluggable backend, one switch:** a `VMBackend` abstraction with two intended implementations — `libvirt-broker` (persistent VMs managed through libvirt/`virsh`, the path implemented today, default `qemu:///system`) and `qemu-direct` (QEMU controlled directly via its QMP protocol, running VMs inside the project container, on the roadmap). The choice depends on whether the host platform exposes hardware virtualisation (`/dev/kvm`, nested virtualisation).

- **In-browser screen:** noVNC over the VNC/RFB protocol, bridged by websockify; one proxy serves all VMs via per-VM tokens.
- **In-guest helper channel:** I use host-only `AF_UNIX` virtio-serial sockets — never on a guest network — built as a *multi-lane* pool (control and file-transfer lanes pinned, heavier calls round-robined across worker lanes) so a long-running call never saturates the channel. I run it at a **single lane** today, because more than two virtio-serial ports destabilise the virtio-win driver on Windows 10/11; widening it there is still open. Frames are newline-delimited JSON-RPC 2.0. (A more compact length-prefixed protocol, *RVMC-WIRE/1* — 4-byte big-endian length prefixes, id-tagged interleaved request/response, per-request timeouts, heartbeats and auto-reconnect — is specified and prototyped as a future upgrade, not yet on the main line.) The Mac OS 9 agent (MacPython 2.3) is instead reached over the guest's emulated network (HTTP/TCP), its IP discovered from the libvirt DHCP lease table.
- **Web routing and access:** Traefik as reverse proxy with per-path prefix stripping; Authentik as single sign-on via forward-auth (resolving identity from the proxied `X-authentik-*` headers). Brokering to institutional federation (eduGAIN and similar) over OIDC/SAML is planned; the current installation wires Authentik OIDC to the chat layer and forward-auth to everything else.
- **Network observation:** packet capture to `pcap`; a per-project caching proxy (*wayback-cache-proxy*) used as an explicit guest proxy, with a per-project certificate authority installed into the guest for HTTPS interception and web-archive replay (*pywb*); stand-in network services for absent servers are planned.
- **AI surface:** tools exposed over the Model Context Protocol (MCP) via the Streamable-HTTP transport, consumed by a web chat interface (Open WebUI), the editor's AI assistant (Continue.dev, inside the browser-based VS Code editor *code-server*), and a researcher's own external MCP agent. rvmc's own **analyst agent** is a server-side tool-calling loop (built with *pydantic-ai*) exposed as one of those MCP tools, so any surface can delegate a whole task to it. Inference is routed through the LiteLLM model gateway by alias, which also meters spend per group (a virtual key per group, mapped from the group's identity).
- **Programmable interface:** a REST API described by an OpenAPI specification, authenticated with a per-project bearer token.

Appendix B — The agentic architecture

I run **two independent agents**, each a tool-calling loop on the same framework (*pydantic-ai* over an OpenAI-compatible model gateway), and I keep them deliberately apart because they do different jobs at different stages.

1. The autopsy analyst (in *artwork-autopsy*) reasons about a work *before* any VM boots. The discipline I hold it to is a hard separation between **extraction** and **judgment**:

- **Deterministic extraction as tools.** A library of self-contained tools *extracts* raw evidence and makes no interpretive claim — file-format identification (*siegfried*, *magika*), native-binary parsing (*lief*), Director/Shockwave, Flash, Java, .NET and Unity decompilation, QuickTime and HyperCard and Mac resource-fork readers, string and metadata scans, entry-point detection, and a network-endpoint probe. I forbid this code from importing the LLM layer; it only surfaces signals.
- **The analyst judges.** A single agentic session drives those tools over the evidence workspace and *reconstructs* what the work is — its crucial parts, its runtime, and which network addresses it assembles at run time versus inert defaults. It **navigates a run-graph** rather than one-shotting a digest: entry-point selection and endpoint discovery happen inside the session, as graph tools it calls, so its conclusions stay grounded in the extracted evidence (the "cartographer" lays a grounded flow over a skeleton). An **independent LLM judge** re-evaluates the result. And for every dead endpoint it finds, the analyst can spawn a **wayback specialist sub-agent** that establishes ground truth from the web archive — it lists the era-matched captures, picks the one that fits the *constructed* request, and fetches the real archived bytes and content-type — so the revival contract

names the actual format a revival must reproduce, not an inference (see Appendix C). This archive-query loop is integral to the analysis, not a runtime add-on. When the analyst is genuinely unsure it can **ask the researcher** — the session blocks, the human answers in the web UI, and it resumes.

- **Telling the artwork apart from its runtime.** A work often ships its whole runtime beside itself — an entire Mac Java runtime of some two and a half thousand files, a bundled Director/Shockwave player, a `node_modules` tree, a Python `site-packages`. Graphed member by member that buries the artwork in thousands of nodes and mis-resolves the entry point *into* the runtime — a demo applet, a dependency library. So the same extract-then-judge split applies to scope. A deterministic detector recognises the unambiguous vendored runtimes from structural signals alone (a JRE/JDK layout, `node_modules`, `site-packages`, a Mac runtime) and **collapses each to a single opaque node** — in the analysis graph only; nothing is deleted, the runtime still ships into the VM. A **scoping specialist sub-agent** judges the harder cases from the evidence — which subtree is the work and which is dependency it merely carries — across ecosystems (JVM/Processing, .NET, JS, Python), and it prefers to keep when unsure: excluding the artwork is worse than graphing a few extra dependency files. This is in active development, generalised from a first Java case; it is why the run-graph can show the work rather than the library it came wrapped in.
- **Why I split it this way.** Deterministic tools flag files fairly and never starve or over-fit to one artwork; the LLM does the constructed, contestable part (artwork vs runtime, live vs dead) where judgment actually belongs. The autopsy's output — a report, a revival runbook, and suggestions — lands on the project drive and is what the workbench acts on.

2. The rvmc analyst agent (in *research-vm-controller*, `rvmc/agent.py`) drives the *revival* itself. Crucially, **I keep this loop in rvmc** — I do not delegate it to a chat product — so rvmc owns the budget and the playbook:

- **A bounded loop.** A hard ceiling on tool-call rounds (a revival is a dozen or two steps) stops a confused model from looping forever and burning the group's AI budget; the limit is enforced as a request cap on the run.
- **A revival playbook as its system prompt.** The prompt encodes the ordering the model would otherwise rediscover every session: read `autopsy_report` → compile `revival_plan` → boot the suggested era machine → launch the work → `verify_running` (guest alive, expected process up, network reached against the endpoints the analysis predicted) → snapshot a known-good state → `record_revival` to close the loop back onto the drive.
- **The same tools as every other surface.** The agent's tools are generated from a **shared capability registry**, so it has exact parity with what a human or a chat model can call — one definition, every consumer (Appendix C). It is itself exposed as a tool (`agent_run`), so a chat assistant or a researcher's own agent can hand it a whole job.
- **Task autonomy, never state autonomy.** Like everything else in the workbench, the agent drives on request and never snapshots over or resets a running machine behind the researcher's back.

The division is the point: the analyst *understands the bytes*; the agent *drives the machine*. Both are built and exercised today; both degrade safely (no key → the analyst falls back to a deterministic evidence-only reasoner; no agent surface → the tools are still callable one at a time).

Appendix C — The MCP tool surface

Every capability the system exposes to a model is a **Model Context Protocol (MCP)** tool, served over the Streamable-HTTP transport. I build **one MCP server per project**, scoped to that project's identity group, currently exposing **over ninety project-scoped tools**. The same server feeds four consumers without duplication: the beside-the-VM chat assistant (Open WebUI), the code editor's assistant (Continue.dev in *code-server*), a researcher's **own external agent**, and rvmc's **own analyst agent** (Appendix B) — all through the one shared capability registry.

The surface, grouped by what it touches:

- **Catalog & lifecycle** — list the master-image catalog and running instances; start, power, and reset VMs; provision, reconcile, and tear down projects; manage members.
- **Analysis** — kick off and read the autopsy (`autopsy_report`, `revival_plan`, `analysis_start/status/report/confirm/cancel`); the `confirm` step is where the human answers the "is this bundled runtime the work or a dependency?" question.
- **The agent** — `agent_run` (hand it a whole revival), `instance_capabilities`, `verify_running`, `record_revival`.
- **Screen & input** — screenshot, send keys, list/set resolution, record the screen.
- **In-guest control** — the helper program's abilities on capable guests: run commands and Python, read/write files and the registry, list and act on windows, send keys and clicks, list processes, and **install software from a virtual CD by clicking through its installer automatically**.
- **Data in and out** — build and attach on-demand CD-ROMs, push uploads into the guest, and the host↔guest data-link bridge (`datalink_build/attach/detach/pull`).
- **Network observation** — start capture; search, tail, and summarise it.
- **Dead-web revival** — point the guest's browser at the per-project caching proxy, install the interception certificate, and toggle archive replay (`wbc_*`).
- **Snapshots & save-points** — VM snapshots and workspace save-points, kept independent on purpose, plus promotion of a verified result.
- **Plugins & tools** — scaffold, read, write, bind, and list conservation plugins and their runtime tools.

Two rules I hold to make this tractable. First, **per-guest-OS capability gating**: the resolved capability set for an instance (module ∩ in-guest agent) is published through the API and MCP, so a connecting agent knows, at discovery time, that (say) Mac OS 9 offers scripting-over-network but not `guest_click` — it gets a clear refusal, not an obscure transport error. Second, **write a tool once**: host actions, guest actions, and a project's own custom analysis tools are all defined in one place and offered everywhere, which is what lets the human, the chat model, the external agent, and the analyst agent share exactly one surface.

A separate, **shared** MCP server — *wayback-mcp* — sits alongside the family as a common side-car over the Internet Archive's Wayback Machine, and it is central to how the analyst reasons about dead endpoints. Four read-only tools do the work: `wayback_snapshots` (the CDX index — every capture of a URL, with timestamp, status and content-type, scoped to the artwork's era), `wayback_available` (the single capture closest to a chosen moment), `wayback_fetch` (the **ground truth** — the archived capture's actual bytes and content-type), and `wayback_prognosis` (a cheap triage that sizes up a whole cluster of a work's dead URLs at once). Resolving a dead endpoint is a search-and-judge loop, not a lookup, so a wayback *specialist sub-agent* drives these tools while the server stays dumb and factual. The autopsy analyst spawns it *during analysis* to establish what each dead endpoint served — which is what turns a "this URL is gone" finding into a concrete revival — and rvmc points at the same container to serve those captures back into a running VM.

Appendix D — The fleet: how the parts compose

vmctl is not one program but a small **fleet of cooperating repositories**, split along the seams that matter for conservation and joined by a shared core. Understanding the split explains the system.

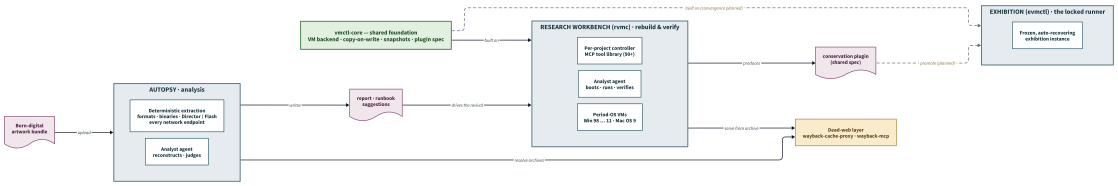

Figure 2: **Figure 2.** The vmctl fleet and the conservation lifecycle. A born-digital work flows autopsy → workbench → exhibition; the research and exhibition controllers share one core (`vmctl-core`) and one plugin spec, and both draw on the dead-web layer. Dashed edges are planned — the workbench-to-exhibition promotion and evmctl's convergence onto the shared core.

The shared core — `vmctl-core`. One package holds what every VM-conservation controller needs: the pluggable VM backend (the `VMBackend` abstraction with its libvirt and direct-QEMU implementations), copy-on-write cloning, snapshots, the in-browser screen plumbing, guest I/O, the master-image catalog, the project/drive model, promotion, and — importantly — the **shared plugin spec**. Both the research and (in future) the exhibition controllers build on it, so a conservation result expressed as a plugin is legible to both.

The parts, by stage:

- **`artwork-autopsy`** — the analysis stage (Appendix B): deterministic extraction plus the analyst, run as its own service over the project drive, with a PostgreSQL-backed job runner.
- **`research-vm-controller` (rvmc)** — the workbench stage (this paper): the per-project controller, the MCP surface (Appendix C), and the analyst agent, all on `vmctl-core`.
- **`exhibition-vm-controller` (evmctl)** — the exhibition stage and the **north star**: the frozen, auto-recovering runner a museum switches on each morning. It is the *original* system rvmc grew out of; folding it onto the shared `vmctl-core` is the convergence still to be done.
- **`wayback-cache-proxy` + `wayback-mcp`** — the dead-web layer: the caching/replay proxy in the VM's request path, and the shared archive-lookup MCP the analyst and workbench both use.

Why I set it up this way. Three separations do real work. *Analysis is separate from the workbench* because understanding the bytes and driving the machine are different tasks with different failure modes — and the autopsy is useful on its own, to anyone, without a VM. *Deterministic extraction is separate from LLM judgment* so the contestable, expensive reasoning is isolated, cheap to audit, and never fabricates evidence. *Research is separate from exhibition* because their requirements are opposite — a research VM must never reset state under the researcher, an exhibition VM must reset the moment anything drifts — yet both can share one core and one plugin format, so a verified result can move from one to the other without re-implementation.

What I still have to do. The through-line I am building toward but have not yet closed: the **automatic hand-off** of a verified result from the workbench to the exhibition runner, and bringing evmctl onto `vmctl-core` so that hand-off is a promotion, not a port. Alongside it: git-backed workspace checkpoints, automatic snapshot housekeeping, the in-container VM backend, reusable stand-in services for absent servers, packaging plugins into self-describing shareable bundles with a registry, and brokering institutional single sign-on (see section 10).

Where I want it to go. I designed the fleet so the end state is not a bigger monolith but a **federation**: cooperating institutes each host the same portable stack, exchange master images and conservation plugins as licensed, self-describing bundles, and move a work along one lifecycle — autopsy → rebuild and verify in the workbench → lock into exhibition. The shared core and the shared plugin spec are what make that lifecycle a continuous path rather than three disconnected tools.

Appendix E — A worked blackbox autopsy

To make the analysis concrete, here is the **actual output** of the autopsy on a single work I ran blind: a Macromedia Director piece distributed as a Mac *StuffIt* archive (2.4 MB), taken as found with no accompanying documentation. I keep the work itself anonymous — what matters here is what the tool found, not whose piece it was. I gave the tool nothing but the archive; everything below is what it reported back.

What deterministic extraction established. Unpacking the archive yielded three components: an entry-point application, a Director cast (`.dcr`), and a resource fork (`.rsrc`). Format identification recognised the classic-Mac *AppleDouble* carrier (via the Mac resource-fork reader) and typed the entry point as a **classic Mac application** (deterministic, high confidence). From that alone the tool fixed the substrate: **base VM Mac OS 9**, runtimes **Classic Mac OS + Director/Shockwave**, work kind *interactive multimedia*. Reading the cast found a **Director "net-Lingo" capability with eight network-operation sites** — but crucially, the static pass reported the destination honestly as **"computed at**

run time — not statically resolved," and flagged the next move: *decompile with ProjectorRays for the full computed paths.* A lesser tool would have guessed or stayed silent; this one named the gap.

What the analyst added. The agentic analyst took that lead, decompiled the (afterburned) cast, and read the Net-Lingo calls. The eight sites resolved to a small set of real endpoints: the plugin-era `download.macromedia.com` / `fpdownload.macromedia.com`, and — the one that matters — the work's own back-end on a university server, reached through `getNetText` / `gotoNetPage`. The analyst's verdict on it is the whole point of the exercise: *if that server is unreachable, the work breaks.* All three hosts are long dead.

Why this is the argument of the paper, in miniature. Nothing in the artwork's *files* announced that it phones home to a university server that stopped answering years ago; the dependency lived inside compiled Lingo, as a string assembled at run time. Only opening the work up surfaced it — and surfacing it is exactly what turns a dead bundle into a conservable one: name the dead host, ask the Wayback Machine — through the shared *wayback-mcp* — what it once served, and replay those captures through the workbench's proxy *before first run*, so the piece has a chance of running again. The autopsy also flagged the **identity-defining fidelity risks** a curator must respect — the piece runs *too fast* on a modern CPU (its pacing is authored into period hardware timing), its fixed-pixel stage must not be rescaled, and its black-and-white dithering is tuned for a CRT — the difference between "it launches" and "it is the work."

References

All online sources below were recorded on 2026-07-22.

Becker, Devin. “Metaphors We Work by: Reframing Digital Objects, Significant Properties, And the Design of Digital Preservation Systems.” *Archivaria* 85 (2018): 6–37. https://archivaria.ca/index.php/archivaria/article/view/13628.

Cochrane, Euan, Klaus Rechert, Seth Anderson, Jessica Meyerson, and Ethan Gates. “Towards a Universal Virtual Interactor (UVI) for Digital Objects.” In “iPRES 2019: 16th International Conference on Digital Preservation.” Special issue, *iPRES 2019: 16th International Conference on Digital Preservation*, 2019, 191–200. https://ipres2019.org/static/pdf/iPres2019_paper_128.pdf.

Delve, Janet, and David Anderson. “Preserving Games Environments via TOTEM, KEEP and Bletchley Park.” In *Preserving Complex Digital Objects*, *Preserving Complex Digital Objects*, edited by Janet Delve and David Anderson. Facet Publishing, 2014. https://wiki.softwareheritage.org/wiki/TOTEM.

A. Depocas, J. Ippolito, C. Jones. *Permanence Through Change: The Variable Media Approach*. Guggenheim Museum Publications, The Daniel Langlois Foundation for Art, Science,, Technology, 2003. https://www.variablemedia.net/pdf/Permanence.pdf.

EMBARK (EU COST Action). “EMBARK: Enabling a Community for Media Art Conservation and Knowledge.” 2025. https://www.dpconline.org/news/dpc-joins-embark.

Ensom, Tom. “Revealing Hidden Processes: Instrumentation and Reverse Engineering in the Conservation of Software-Based Art.” *The Electronic Media Review* 5 (2018). https://resources.culturalheritage.org/emg-review/volume-5-2017-2018/ensom/.

Falcão, Patrícia, Annet Dekker, and Pip Laurenson. “An Exploration of Significance, Dependency, And Virtualization in the Conservation of Software-Based Artworks.” *The Electronic Media Review* 4 (2016). https://resources.culturalheritage.org/emg-review/volume-4-2015-2016/falcao/.

Kreymer, Ilya. “A Prototype of Automated Web Archiving, Emulation and Server Preservation.” August 28, 2018. https://blog.conifer.rhizome.org/2018/08/28/automation-emulation-server-preserve.html.

Rechert, Klaus, and Rafael Gieschke. “Preserving Secondary Knowledge — Using Language Models for Software Preservation.” *International Journal of Digital Curation* 18, no. 1 (2024). https://doi.org/10.2218/ijdc.v18i1.930.

Rechert, Klaus, Dragan Espenschied, Rafael Gieschke, and Wendy Hagenmaier. “Preserving Users' Knowledge of Contemporary and Legacy Computer Systems: Automated Analysis of Recorded User Activity.” In “iPRES 2024: International Conference on Digital Preservation.” Special issue, *iPRES 2024: International Conference on Digital Preservation*, 2024. https://doi.org/10.21428/5676bf2d.ff6a2943.

Satyanarayanan, Mahadev, Gloriana St. Clair, Benjamin Gilbert, et al. *One-Click Time Travel*. Technical Report CMU-CS-15-115. 2015. https://olivearchive.org/static/documents/CMU-CS-15-115.pdf.