Tool Boundaries for Containerized Agents

> A practical architecture for deciding what belongs in model backends, project toolchain containers, and external authority gateways.

This is a living white paper. It is maintained as agent architectures, research, protocols, and operating patterns evolve.

Executive summary

Containerized AI agents need tools. The requirement is simple in theory, but it becomes a design problem once an agent is expected to maintain real software across many repositories, languages, clouds, and operating environments.

A desktop agent inherits the developer workstation. If the human has `git`, `go`, Python, Rust, Node, `kubectl`, Helm, cloud CLIs, browser access, local credentials, and package caches, the agent can often use them directly. That is convenient, but implicit: the agent gets whatever the host happens to provide.

A containerized agent has no such default. Every capability must be deliberately supplied. The platform has to decide where compilers live, where CLIs live, where credentials live, where source code is mounted, how commands are audited, and how one model backend can work on projects that need very different development environments.

Witwave currently runs three real model backends: Claude, OpenAI, and Gemini. Each backend is its own image and its own A2A server. The platform also has a shared `backend-base` image that gives those backends a pinned baseline of common command-line tools: Go, Node, `kubectl`, `ww`, `gh`, Helm, ruff, shellcheck, hadolint, gitleaks, trivy, and related analysis and test tooling.

This paper uses Witwave as the concrete reference implementation, but the boundary applies more broadly to any long-running, containerized agent platform. The details will vary by runtime, but the architectural pressure is the same: model execution, local project execution, and external authority should not collapse into one container just because the agent needs all three.

That baseline is useful. It also raises the next design question: where should project-specific execution live when a workspace needs tools beyond the shared platform baseline?

A better boundary is:

- **Backend containers** run model-specific agent runtimes and protocol surfaces.

- **Toolchain containers** provide project-specific local execution environments.
- **MCP and other gateways** expose standardized tool calls into those environments or into external systems.

MCP is a practical protocol for this boundary. A toolchain sidecar can expose an MCP server over localhost, and the backend can call that server using the MCP support it already has. The important distinction is not "MCP versus native." It is whether the language-specific runtime lives inside the backend image or inside a dedicated execution container.

This paper argues for a hybrid model: backends run model runtimes, toolchain containers run project-local tools, and MCP or similar gateways mediate structured calls into toolchains and external systems. The goal is not to remove native tools or avoid MCP; it is to make tool placement explicit.

The immediate question

The design question is:

> How should a containerized AI backend execute project-specific tools without baking every possible language and workflow into every backend image?

The problem is already concrete. Witwave itself is a mixed Go and Python codebase with Helm charts, Kubernetes operator code, static-site content, Dockerfiles, SOPS-encrypted secrets, release automation, and agent configuration. The current backend images carry enough tooling to work on this repository. That is reasonable for Witwave today.

Witwave is meant to run agents against more than this repository. Another workspace may need Rust. Another may need Node. Another may need Java. Another may need AWS account tooling, Terraform, Foundry, Solana, mobile tooling, or a private compiler. Those tools need a deliberate home.

The better framing is not "MCP or native tools?" It is:

> Which capabilities belong in the model backend, which belong in a project execution environment, and which belong behind an external authority gateway?

That framing gives the platform a more useful design vocabulary.

Terminology

Backend

A backend is the model execution container. In Witwave today, the production backends are Claude, OpenAI, and Gemini. Each backend is a standalone A2A server. Each owns its model SDK integration, session handling, conversation logs, memory, metrics, protected inspection endpoints, and provider-specific runtime behavior.

The backend receives identity and behavior through mounted files:

- `CLAUDE.md` for Claude.
- `AGENTS.md` for OpenAI.
- `GEMINI.md` for Gemini.

The backend may expose provider-native tools. For example, Claude can use Claude Code-style tools such as read/search, Bash, edit/write, and MCP depending on configured permissions. OpenAI can expose a shell tool through its Agents SDK integration. Gemini participates in the same backend layout and MCP configuration posture, though some lower-level tool-loop interposition remains less mature than Claude and OpenAI.

The backend should know how to call tools. It should not have to contain every possible project tool.

Native tool

A native tool is directly available inside the container where the model backend is running. For a desktop agent, native tools are host tools. For a containerized agent, native tools are binaries installed in the backend image or mounted into that backend container.

Examples:

- `git`
- `gh`
- `ww`
- `kubectl`
- `helm`
- `go`
- `python`
- `pytest`
- `ruff`
- `cargo`

- `rustfmt`
- `terraform`
- `make`

Native tools are direct and familiar. They are also the closest coupling between model runtime and project execution.

MCP-mediated tool

An MCP-mediated tool is exposed by an MCP server and called through the Model Context Protocol. The server may run as a cluster-shared service, a same-pod sidecar, or an external endpoint. What matters is that the model sees a structured tool surface instead of arbitrary ambient shell access.

Examples:

- Kubernetes inspection or remediation tools.
- Helm release-management tools.
- Prometheus query tools.
- GitHub issue, pull request, or discussion tools.
- AWS account inspection tools.
- PagerDuty or incident-management tools.
- Toolchain execution tools such as `rust.cargo_test` or `python.pytest`.

MCP is strongest when the tool surface is intentionally described, bounded, observable, and policy-aware. That applies to external systems, and it can also apply to local toolchain containers.

Toolchain container

A toolchain container is a project-specific or language-specific execution environment mounted beside the backend. It contains compilers, interpreters, package managers, linters, test runners, project utilities, caches, and related local execution dependencies.

Examples:

- `toolchain-go-python`
- `toolchain-rust`
- `toolchain-node`
- `toolchain-terraform`
- `toolchain-witwave`

The backend cannot run a process inside a sibling container by default. Kubernetes containers in one pod share network and volumes, but they do not share process environments. Therefore the toolchain container needs a deliberate execution surface: usually an HTTP API, an MCP server, gRPC, or a small local daemon.

External authority gateway

An external authority gateway is a service that holds or uses credentials for systems outside the workspace: Kubernetes, AWS, GCP, GitHub, observability systems, ticketing systems, incident-management systems, or secrets systems.

A gateway may also expose MCP. The defining feature is not the protocol; it is the authority boundary.

A Rust toolchain and an AWS gateway can both expose MCP tools. The Rust toolchain boundary is the containerized project execution environment. The AWS gateway boundary is credentials, account scope, audit, and blast radius.

What we have today

Witwave already has a working tool model: backend-native tools, MCP tools, shared images, git-backed configuration, workspace volumes, and Kubernetes-scoped authority. The remaining gap is narrower: there is no first-class home for project-specific execution environments.

Runtime shape

A named Witwave agent is deployed as a pod-shaped unit. The named agent is the operational boundary, and its containers cooperate around a shared runtime:

- A **harness** container receives A2A traffic, schedules heartbeats, runs jobs/tasks/triggers/continuations, and routes work to a backend according to `.witwave/backend.yaml`.
- One or more **backend** containers run model-specific A2A servers. The common production shape can include Claude, OpenAI, and Gemini sidecars for the same named agent.
- Optional **git-sync** sidecars materialize repo content into the pod.
- Optional **MCP tools** run as separate deployments/services, rendered by the chart or operator.
- `WitwaveWorkspace` references mount shared volumes and projected config into participating containers.

The repo remains the source of truth. Agent identity, routing, prompts, schedules, backend instructions, settings, MCP configuration, skills, docs, and website content live in git. At runtime, git-sync and workspace mounts project that repo-managed state into the pod at stable paths.

Backend images and tool surfaces

Witwave currently maintains separate backend images for:

- Claude.
- OpenAI.
- Gemini.

Those images share `images/backend-base/`, published as `ghcr.io/witwave-ai/images/backend-base:<version>`. The base image includes common CLIs, runtimes, and analyzers such as Go, Node, `kubectl`, `ww`, `gh`, Helm, ruff, shellcheck, hadolint, gitleaks, trivy, and test tooling.

That base gives the backends parity for common platform work and pins shared tool versions once. It is image composition, not authority: installing `kubectl` does not grant cluster access. In-cluster Kubernetes access is handled separately through `WitwaveAgent.spec.kubernetesApiAccess` or explicit ServiceAccount/RBAC configuration.

Each backend has its own tool path:

- **Claude** uses Claude Code-style permissions. Read/search tools are allowed by default; Bash, Write/Edit, and WebFetch require explicit enablement through `ALLOWED_TOOLS` or `.claude/settings.json` `permissions.allow`. Claude also reads `.claude/mcp.json` through `MCP_CONFIG_PATH`, validates and hot-reloads MCP configuration, and applies the shared stdio MCP command allowlist.
- **OpenAI** reads `.openai/config.toml` for built-in tool flags and `.openai/mcp.json` for MCP servers. Its `ShellTool` runs local shell commands inside the OpenAI backend container with baseline denial rules, audit logging, timeouts, trace instrumentation, and the shared MCP command/argument safety checks.
- **Gemini** follows the same backend layout: mounted identity document, memory/log paths, MCP configuration shape, metrics, and protected backend endpoints. Its local tool loop is less mature, but the goal is to keep the same tool-surface direction across backends.

Today, backend-native execution still runs inside the backend container. If Claude Bash or OpenAI `ShellTool` runs `go test`, it uses the tools installed in that backend image. MCP can call external services or future sidecars, but Witwave does not yet generate a dedicated project toolchain sidecar for local execution.

MCP components

Witwave currently ships MCP components under `tools/`:

- `mcp-kubernetes` for Kubernetes API access.
- `mcp-helm` for Helm release management.
- `mcp-prometheus` for Prometheus queries.

They run long-lived FastMCP HTTP servers on port `8000`, expose `/health`, enforce bearer-token auth unless explicitly disabled, and are consumed by backend `mcp.json` entries. Chart-rendered MCP tools are cluster services such as `http://<release>-mcp-kubernetes:8000`, not binaries inside a backend container.

Stdio MCP entries are guarded as well. The shared `mcp_command_allowlist` restricts accepted commands and rejects unsafe interpreter forms such as inline code, stdin scripts, unsafe `uv` / `uvx` patterns, or positional scripts outside allowed paths. This gives Witwave useful MCP safety machinery, but the current MCP tools mostly expose cluster and observability gateways. They do not provide local project execution for commands such as `cargo test`, `npm test`, or `terraform validate`.

Shared filesystem and authority boundaries

Witwave relies on repo-managed files and stable mounted paths:

- `.witwave/` contains harness runtime config: `backend.yaml`, `HEARTBEAT.md`, jobs, tasks, triggers, continuations, webhooks, and the public agent card.
- `.claude/`, `.openai/`, and `.gemini/` contain backend-specific identity and tool config.
- `.agents/` stores self/test team definitions and SOPS-encrypted secret mirrors.
- `WitwaveWorkspace` provisions shared volumes and stamps shared config files or existing Secrets onto participating agents.
- Runtime memory, logs, task-store state, and conversation state persist through backend/harness storage paths.

Kubernetes authority is also explicit. When `spec.kubernetesApiAccess` is enabled, the operator creates a per-agent ServiceAccount, namespace-scoped Role, and RoleBinding. Current presets distinguish read-only inspection from bounded namespace-write remediation. This separates image composition from authority, a pattern future toolchains should preserve.

What is missing

The missing layer is not MCP support, shell support, or shared storage. Those already exist in different forms. The missing layer is a first-class execution environment abstraction: this workspace needs these project tools, mounted here, exposed through this safe tool surface, with this policy.

Today there is no:

- `toolchains:` block on `WitwaveAgent` or `WitwaveWorkspace`.
- Operator or chart renderer for toolchain sidecars.
- Standard toolchain image contract.
- Generated backend MCP config pointing to local toolchain sidecars.
- Common `toolchaind` or structured MCP server for project-local execution.

- Policy model for command, cwd, timeout, output, environment, network, and secrets at the toolchain boundary.
- Trace model that distinguishes local project execution from external authority calls.
- Routing model that explains why `cargo test` ran in the Rust toolchain and `pytest` ran in the Python toolchain.

That is the gap: Witwave can already give agents tools, but it does not yet give project-specific execution a clear architectural home.

Why backend images should not become universal toolboxes

The backend images should remain small, generic, and stable. Their job is to run model backends reliably. Claude needs the Claude runtime. OpenAI needs the OpenAI Agents SDK runtime. Gemini needs the Google Gemini runtime. Each backend already has provider-specific dependencies, configuration, tool behavior, metrics, and failure modes.

If every project tool lives inside those backends, every new capability becomes backend-image work. A new language, a new linter suite, a new build system, or a new integration must be evaluated against Claude, OpenAI, and Gemini even when the tool has nothing to do with the model provider.

The shared `backend-base` image is the right place for common platform utilities. It removes duplicated setup across the three backend Dockerfiles and keeps baseline versions pinned in one place. It should not become the default landing zone for project-specific runtimes. Anything added to `backend-base` becomes part of every backend image that inherits from it.

That creates the wrong maintenance matrix:

```
backend type x project toolchain x version x security posture
```

The failure modes are predictable:

- Backend images grow large.
- Build times increase.
- CVE surface expands.
- Toolchain version drift becomes harder to reason about.
- Provider runtime changes are coupled to project language changes.
- Project portability suffers because each repo wants a custom backend image.

- Ownership becomes unclear: should the Claude backend image own Rust versioning?

Dedicated toolchain containers break that multiplier. Adding Rust support means creating or updating one Rust toolchain container. Adding a linter suite means updating the relevant project toolchain. Adding an external integration means creating a gateway or controlled tool surface with its own policy and release cadence.

The backend should know how to call tools. It should not have to contain every tool.

The sidecar toolchain model

The proposed model places project execution environments beside the backend, not inside it.

```
agent pod
├── harness
├── claude-backend
├── openai-backend
├── gemini-backend
├── toolchain-go-python
├── toolchain-rust
└── shared workspace volume
```

Each toolchain container provides:

- A project-specific or language-specific image.
- The shared workspace mounted at a known path.
- A small local execution service.
- A localhost port inside the pod.
- Structured tools or tightly controlled command execution.
- Command, cwd, timeout, output, and environment policy.
- Audit logs and metrics.
- Resource requests and limits independent of the backend.

The backend calls the toolchain service. It does not directly spawn toolchain processes.

For example, a Rust toolchain sidecar might expose an MCP tool equivalent to:

```
{
  "tool": "rust.cargo_test",
  "arguments": {
    "cwd": "/workspaces/witwave/source",
    "package": "operator",
    "timeoutSeconds": 300
  }
}
```

Or it might expose a more generic but constrained execution tool:

```
{
  "tool": "toolchain.exec_allowed",
  "arguments": {
    "command": ["cargo", "test"],
    "cwd": "/workspaces/witwave/source",
    "timeoutSeconds": 300
  }
}
```

The toolchain container runs the command inside the Rust environment and returns structured output:

```
{
  "exitCode": 0,
  "stdout": "...",
  "stderr": "",
  "durationMs": 18420,
  "truncated": false
}
```

This keeps the Rust compiler in the Rust environment, the LLM runtime in the backend environment, and the source tree as the shared contract between them.

How the backend would execute toolchain work

Several integration paths are possible. They can coexist, but they are not equally desirable.

Option 1: Toolchain sidecars expose MCP

This is the most practical first implementation path.

The backends already know how to consume MCP configuration. Claude, OpenAI, and Gemini use the same broad `mcp.json` shape. A toolchain sidecar can run an MCP server over HTTP on localhost; the chart/operator layer can render backend MCP entries that point at those local sidecars.

Example generated backend MCP configuration:

```
{
  "mcpServers": {
    "toolchain-go-python": {
      "url": "http://localhost:8701/mcp"
    },
    "toolchain-rust": {
      "url": "http://localhost:8702/mcp"
    }
  }
}
```

The Rust toolchain MCP server might expose:

```
rust.cargo_check
rust.cargo_test
rust.rustfmt_check
rust.clippy
toolchain.exec_allowed
```

The Go/Python toolchain MCP server might expose:

```
go.test
go.vet
python.pytest
python.ruff_check
python.ruff_format_check
toolchain.exec_allowed
```

This is not a compromise or a misuse of MCP. MCP can be the standardized communication protocol between the backend and the toolchain container. The key boundary is the container boundary: language-specific tools live in a dedicated execution environment rather than in the backend image.

Benefits:

- Reuses existing backend MCP support.
- Avoids immediate per-backend native tool implementation.
- Gives each toolchain a described model-facing tool surface.
- Allows multiple named toolchains in one pod.
- Makes tool availability explicit to the model.
- Fits existing MCP auth, body-cap, metrics, audit, and reload patterns.

Risks and controls:

- Generic `exec` remains powerful and needs strict policy.
- Tool descriptions must be precise and safe.
- CWD must be restricted to approved workspace paths.
- Commands and arguments need allowlists or structured wrappers.
- Timeouts and output caps are mandatory.
- Toolchain names and descriptions must be clear enough that the model can choose correctly.

This is the best initial path.

Option 2: Backend-native `run_toolchain` tool

The backend could expose a provider-native tool named `run_toolchain` and call a local toolchain HTTP API directly.

Example:

```
{
  "toolchain": "rust",
  "command": ["cargo", "test"],
  "cwd": "/workspaces/witwave/source",
  "timeoutSeconds": 300
}
```

Benefits:

- Clear product abstraction.
- Backend traces can represent toolchain calls consistently.
- A central backend policy layer can mediate calls.
- The toolchain contract is not dependent on MCP.

Costs:

- Requires implementation in each backend.
- Claude, OpenAI, and Gemini have different native tool APIs.
- Slower to ship.
- Adds another protocol surface to maintain.

This may become attractive later, but it is not the simplest place to start.

Option 3: Backend uses `kubectl exec` into a sibling container

This shortcut is tempting:

```
kubectl exec <pod> -c toolchain-rust -- cargo test
```

It should remain a debugging technique, not a platform primitive.

It requires the backend to have `pods/exec` permission. It turns Kubernetes into the command-execution API. It is harder to audit as an agent tool action. It couples local project execution to Kubernetes API authority even though the containers already share a pod. It also encourages granting broad Kubernetes capabilities to a model-facing backend.

Option 4: Project-specific backend images

A project can still build a custom backend image:

```
FROM ghcr.io/witwave-ai/images/claude:0.27.1

RUN install-rust-toolchain
RUN install-project-tools
```

This is simple and sometimes useful, but it should remain an escape hatch. It repeats work across backend types, bloats backend images, and couples project toolchains to provider runtimes.

Option 5: Ephemeral runner jobs

Heavy or risky execution can move into short-lived Kubernetes Jobs:

```
run cargo test in image ghcr.io/org/project-rust-toolchain:sha
mount workspace snapshot
return logs and exit code
```

This is useful for expensive builds, matrix tests, integration tests, or untrusted workloads. It provides stronger isolation and a cleaner resource lifecycle, but it is slower and more complex than a sidecar. It should complement the sidecar model rather than replace it at first.

MCP as protocol, toolchain as boundary

MCP is a protocol. Toolchain is an architectural role.

A Rust toolchain, a Go/Python toolchain, a Prometheus gateway, and an AWS gateway can all expose MCP tools. The question is not whether local toolchains may use MCP; they can. The question is whether the platform preserves the right boundary behind the protocol.

For toolchains, the boundary is local execution:

- Workspace-mounted source code.
- Language runtimes.
- Compilers.
- Linters.
- Test runners.

- Package caches.
- Local build scripts.

For external systems, the boundary is authority:

- Credentials.
- Account scope.
- Cluster scope.
- Read/write posture.
- Audit.
- Blast radius.

The same protocol can serve both layers. The layer still matters.

Capability	Primary concern	Better conceptual bucket
`cargo test`	Local project execution	Toolchain
`go test ./...`	Local project execution	Toolchain
`ruff format`	Local project mutation	Toolchain
`make test`	Local project execution	Toolchain
`kubectl get pods`	Cluster authority	Native tool or MCP gateway
`helm template`	Local chart rendering	Toolchain
`helm upgrade`	Cluster mutation	MCP gateway or controlled CLI
`aws sts assume-role`	Cloud authority	MCP/gateway
`aws s3 ls prod-bucket`	Cloud authority/data	MCP/gateway
`gh issue create`	Source-control authority	MCP/gateway or native CLI
`prometheus query`	Observability access	MCP/gateway

The claim is modest: use MCP where it helps, but do not let the protocol obscure whether a tool represents local execution or external authority.

Recommended first design

Start with a deliberately small design:

> Add named toolchain sidecars that expose structured MCP tools over localhost.

The platform primitive should be `toolchains`; the first transport can be MCP.

Example future agent spec:

```yaml
spec:
  toolchains:
    - name: go-python
      image:
        repository: ghcr.io/witwave-ai/toolchains/go-python
        tag: "0.1.0"
      port: 8701
      mountPath: /workspaces/witwave/source
      tools:
        mode: structured
      allowedCommands:
        - go
        - python
        - pytest
        - ruff
        - make

    - name: rust
      image:
        repository: ghcr.io/witwave-ai/toolchains/rust
        tag: "0.1.0"
      port: 8702
      mountPath: /workspaces/witwave/source
      tools:
        mode: structured
      allowedCommands:
        - cargo
        - rustc
        - rustfmt
        - clippy
```

The operator would render:

- One sidecar per toolchain.
- Workspace mounts into each toolchain sidecar.
- Backend MCP config entries for local toolchain URLs.

- Per-toolchain resource requests and limits.
- Per-toolchain security context.
- Optional environment variables describing toolchain names and URLs.
- Optional network-policy controls.

Each sidecar would provide:

- `/health`.
- `/metrics`.
- An MCP endpoint.
- Structured tools.
- Optional constrained `exec_allowed`.
- CWD restriction.
- Timeout enforcement.
- Output caps.
- Audit logs.

The backend would not need Rust in its own filesystem. It would only need to know that a `toolchain-rust` MCP server exists and exposes safe Rust tools.

Multiple execution environments

Multiple toolchains should be supported, but bounded.

An agent pod can reasonably run a small number of named toolchain containers:

```
toolchain-go-python
toolchain-rust
toolchain-node
toolchain-terraform
```

It should not run hundreds. Each toolchain brings image pulls, resource requests, patch cadence, security posture, logs, metrics, and operational overhead.

The design should optimize for:

- One default project toolchain.
- Optional language-specific toolchains for heavy runtimes.
- Optional high-risk toolchains with tighter network or secret posture.
- Optional runner jobs for expensive matrix work.

Toolchain selection should start explicit:

```
Use the Rust toolchain to run cargo test.
Use the Go/Python toolchain to run ruff and pytest.
Use the Terraform toolchain to run terraform fmt and validate.
```

Later, the platform can add routing hints:

```
        handles:
          files:
            - Cargo.toml
            - "*.rs"
          commands:
            - cargo
            - rustc
            - rustfmt
            - clippy
```

Automatic selection should be explainable and traceable. If an agent runs `cargo test`, the trace should show:

```
toolchain selected: rust
selection reason: command cargo matched toolchain rust handles.commands
```

Hidden routing should wait until traces show the common patterns. If the model asks for `make test`, the platform should not silently choose between Go/Python, Rust, Node, or project-default toolchains without recording an explanation.

Security posture

Toolchain execution is powerful: local code execution against a shared workspace. Treat it as a high-trust but bounded capability.

Minimum controls:

- **No cluster credentials by default.** Toolchain containers should not automatically receive Kubernetes service account tokens.
- **No cloud credentials by default.** AWS, GCP, Azure, GitHub, and other external authority should be mediated separately unless explicitly required.
- **CWD restriction.** Commands should run only inside approved workspace paths.
- **Structured tools first.** Prefer `rust.cargo_test` or `python.pytest` over a generic shell when possible.
- **Command allowlist.** If generic exec exists, restrict approved binaries.
- **Argument policy.** Interpreters and package runners need special handling because `python -c`, `node -e`, `bash -c`, `uvx`, `npx`, and similar forms can defeat naive command allowlists.
- **Timeouts.** Every execution needs a bounded timeout.
- **Output caps.** Large output should be truncated with clear metadata.
- **Audit logs.** Record toolchain name, backend, session hash, command or structured tool, cwd, exit code, duration, truncation, and policy decisions.
- **Resource limits.** Toolchains should have CPU and memory requests/limits independent of the backend.
- **Network posture.** Some toolchains need package download access; others should be egress-restricted.
- **Secret posture.** Secrets required for external systems should not be casually mounted into local build toolchains.

The platform should assume prompt injection can cause a backend to call available tools. The response should not be to withhold all tools; it should be to place tools behind boundaries that match their risk.

Relationship to dev containers

The Dev Container ecosystem is relevant because many repositories already describe their development environments in `.devcontainer/devcontainer.json`.

A future Witwave toolchain design could support:

```
toolchains:
  - name: default
    devcontainer:
      path: .devcontainer/devcontainer.json
```

This would let projects reuse an existing development-container definition instead of writing a Witwave-specific image from scratch.

Dev containers are not a complete solution. The platform still needs:

- A backend-to-toolchain call path.
- A model-facing tool surface.
- Audit logs.
- Timeouts and output caps.
- Secret and network policy.
- Support for more than one execution environment.

Dev containers can help define the image. They do not define the agent execution contract.

Relationship to MCP gateways

Toolchains should sit beside MCP gateways, not replace them.

A possible deployment could look like this:

```
Local project execution:
  toolchain-go-python
  toolchain-rust
  toolchain-node

External authority:
  mcp-kubernetes
  mcp-helm
  mcp-prometheus
  mcp-aws-prod-readonly
  mcp-github-discussions
```

Some tools have both local and authority-bearing modes:

Kubernetes:

- `kubectl` in a backend or toolchain container is useful for diagnostics and parity with human workflows.
- Production cluster access is safer behind scoped identity, RBAC, and often an MCP gateway.

Helm:

- `helm template` and `helm lint` are local chart work and fit a toolchain.
- `helm upgrade` against a live cluster is external authority and may belong behind a gateway or a tightly controlled native path.

AWS:

- Native `aws` CLI can work for one account with one scoped identity.
- Multi-account role switching is easier to audit as a gateway surface scoped by account, environment, and permission tier.

Prometheus:

- Prometheus is mostly observability access, so MCP is a natural fit.
- The value comes less from where the client binary lives and more from scoping query access, limiting response size, and auditing what the agent asked.

Implementation stages

Stage 1: Keep the paper as a design record

Name the boundaries clearly:

- Backend runtime.
- Toolchain execution.
- External authority gateway.

Use this paper to keep the boundary clear whenever a new language or CLI appears.

Stage 2: Build one local toolchain sidecar

Build a simple `toolchain-go-python` sidecar for the current repo.

It should:

- Run as a sidecar.
- Mount the same source/workspace volume as the backend.
- Expose `/health` and `/metrics`.
- Expose an MCP server over localhost.
- Provide structured tools for the current repo's common checks.
- Record audit logs.
- Enforce timeouts and output caps.

Initial tools could be:

```
go_test
go_test_package
python_pytest
python_ruff_check
python_ruff_format_check
repo_command_allowed
```

Avoid starting with a wide-open shell.

Stage 3: Add chart/operator support

Add a `toolchains` block to the chart and operator.

Minimum fields:

- name
- image
- port
- workspace mount path
- resources
- env
- allowed commands or structured mode
- enabled flag

The operator should render sidecars and backend MCP config entries.

Stage 4: Add a second language toolchain

Add `toolchain-rust`.

This proves the design is not hardcoded to the current Go/Python repository.

Initial tools:

```
cargo_check
cargo_test
rustfmt_check
clippy
```

Stage 5: Add routing hints

Add optional metadata:

```
        handles:
          commands: ["cargo", "rustc", "rustfmt", "clippy"]
          files: ["Cargo.toml", "*.rs"]
```

Expose hints to the model and traces first. Add automatic routing only after usage patterns are clear.

Stage 6: Add runner jobs for heavy execution

For expensive builds, integration tests, matrix runs, or untrusted code, use short-lived runner pods or jobs rather than long-running sidecars.

Open design questions

Should toolchains be per-agent or per-workspace?

Per-agent is easier to render because agents already own their pods.

Per-workspace is conceptually cleaner because toolchains are project-specific, not personality-specific. Multiple agents bound to the same workspace probably want the same execution environments.

A practical path is to support per-agent toolchains first and leave room for workspace-level defaults later.

Should package caches be shared?

Probably, eventually. Language package caches are expensive to rebuild.

Shared caches introduce state, invalidation, poisoning, and storage concerns. The first version can tolerate slower execution in exchange for simpler semantics.

Should toolchains have network access?

Some must. Package managers need network access unless dependencies are vendored or cached.

Network access changes the risk model. A toolchain running project scripts with broad egress is a larger attack surface than one limited to local compilation. Network policy should become part of the toolchain spec.

How generic should the execution tool be?

The safest first tools are structured: `cargo_test`, `go_test`, `python_pytest`, `ruff_check`.

A generic `exec_allowed` tool may still be necessary, but it should be treated as a controlled escape hatch with strict command, cwd, argument, timeout, output, and audit policy.

How much should the model choose automatically?

Not much at first.

The model can choose from clearly named tools. The platform can later add routing hints and defaults. Hidden automatic routing should wait until traces show common patterns.

Evidence and boundaries

This paper is an architectural design argument, not a benchmark result. It makes a narrow claim: containerized agent platforms need a stable boundary between model runtimes, project-local execution, and external authority.

The recommendation is grounded in several practical constraints:

- Kubernetes pods are designed for co-located containers that can share storage and network resources, making sidecars a natural fit for tightly coupled support services.
- MCP provides a standard protocol surface for tool calls, but the protocol does not decide where authority or execution should live.
- Dev containers show that development environments can be described as reusable containerized tool surfaces, but they do not define an agent execution contract by themselves.
- Kubernetes RBAC and ServiceAccounts separate having a CLI installed from having permission to use cluster authority.
- OWASP and NIST guidance both point toward explicit tool permissions, observability, auditability, and least privilege when agents can call tools or execute commands.

The paper intentionally does not claim that every project needs multiple toolchain containers, that every command should be MCP-mediated, or that native tools should disappear. It argues for clearer placement. Common platform utilities can belong in a shared backend base. Project-specific build and test environments should have a first-class home. External systems with credentials and blast radius should be treated as authority boundaries.

Conclusion

The recommendation is straightforward: make toolchains a first-class architectural layer.

Witwave should treat model runtime, project-local execution, and external authority as separate responsibilities. The repo and workspace mounts remain the shared source of truth. External systems remain behind explicit authority gateways. Project-local execution belongs between those layers, in dedicated toolchain containers that can be mounted, governed, traced, and replaced independently.

MCP fits this design well. It can be the standard protocol a backend uses to call a toolchain sidecar, just as it can be the standard protocol for calling Kubernetes, Helm, Prometheus, GitHub, or cloud gateways. The protocol is not the boundary. The container, credential, and authority model is the boundary.

That separation lets Witwave grow one layer at a time. Toolchains evolve with projects. Gateways carry external authority. Backends remain focused on model work. The agent gets the tools it needs, and the platform keeps a clear answer to where those tools belong.

Sources and further reading

- Model Context Protocol. **Base Protocol Overview**. Useful for grounding MCP as a JSON-RPC protocol surface between clients and servers.
- Kubernetes. **Pods**. Useful for the co-located container, shared storage, and shared network assumptions behind sidecar toolchains.
- Kubernetes. **Service Accounts** and **Using RBAC Authorization**. Useful for separating installed tools from Kubernetes API authority.
- Development Containers. **Development Containers**. Useful for the idea that development environments can be represented as reusable container definitions.
- Open Container Initiative. **Open Container Initiative**. Useful background for the image, runtime, and distribution standards behind portable containerized execution.
- OWASP. **MCP Top 10**. Useful for MCP-specific risks around secret exposure, scope creep, command execution, authentication, telemetry, shadow servers, and context over-sharing.
- NIST, 2025. **Lessons Learned from the Consortium: Tool Use in Agent Systems**. Useful for reasoning about tool functionality, access patterns, risk, reliability, monitoring, and autonomy.