

Ethgent: Fitting an Agent to a System by Separating Agent, Connection and Workflow Factory

Ahmet Rifat Öztürk Yağız Ekrem Dalar Ömer Faruk Aksoy
ahmetrifat@ethosoft.org ekrem@ethosoft.org omer@ethosoft.org
Nedim Mutlu Sezer Feyzi Arda Salihoğlu
nedim@ethosoft.org feyziarda.salihoglu@ethosoft.org
Ethosoft

Preprint, October 2026; revised 16 October 2026. Not peer reviewed.

Licensed under [CC BY 4.0](#).

Abstract

A company that wants a language-model agent to work for it has to put that agent inside its own backend service, join it to the application and the outside world, and shape its orchestration to the company’s own work. This paper is about that relationship between the agent, the backend service and the system around it. A model becomes useful there only when it is *fitted*: given the right instructions and tools, a way to read what happens, a way to act on it, and limits on what it may do. Fitting should not mean building a new agent for every company or domain. We describe a small set of open-source components that separate these concerns instead of fusing them in one process: a customisable agent whose agents and tools are configuration (Ethgent), a connection layer and open pool of workers, connectors and contracts (Marketing Agent Assets), a workflow factory that turns capabilities into deterministic, exportable tools (MarketingStudio), and a worked example that runs the combination on one machine (MarketingPool). Our working domain is marketing, but the design is domain-neutral: what changes between domains is the instructions, the tools and the connections, and we say precisely which parts those are. We built no second domain, so that is an argument from the design, not a result. The aim is an open contribution system: through GitHub and a common export format, different people can add an agent, a tool, a skill or a worker, and an organisation can assemble them into an extension that runs inside its own system. Credentials stay with workers; anything that changes the outside world is approved by the host, not by the model; every component works without the others and they meet through plain files and small contracts. We report what we ran: a workflow written in the factory, exported, installed and called by an agent backed by a hosted model, including one that needs approval, one that takes a file and one that needs a package, two design faults that this exercise exposed and fixed, and a measured comparison (a three-step task and an eight-step chain, two models of one vendor, 15 to 20 runs per cell): giving the agent a workflow as one tool instead of the same steps as separate tools cut tokens by 10 to 40 percent on three steps and by a factor of about two on eight, with no statistically detectable difference in reliability. We also ran the agent as a worker inside the backend service of a real application of ours: the application’s contract checker passed (19 of 19), the agent saw only the lessons the application approves, and a database role limited it to its own queue. We do not report an end-to-end deployment against real platforms, and we say where the evidence ends.

1 Introduction

A language model on its own can talk. Put inside a system, it has to do four further things: know what it is for (instructions), have something to do it with (tools), see what is happening

	The company needs to	It writes or sets	Tested here
1	Let the agent see what the application approves, and nothing else	A small endpoint contract on the application, or a worker that holds the platform credentials (Section 2)	Yes, with our own application: the contract checker passed 19 of 19 and a private lesson never appeared (Section 8)
2	Run the agent inside its own service boundary	The agent as a queue worker next to the backend, with a database role that can touch its queue and nothing else	Yes, in one Docker Compose project (Section 8)
3	Shape the orchestration to its work	Which sub-agents are on; each one’s prompt, model and tools; the orchestrator’s prompt; packs pulled from GitHub (Sections 2.2, 2.3, 5)	Packs and a workflow tool installed and called (Section 6); changing the orchestrator’s prompt was not part of our tests
4	Keep authority with its own people	Which tools declare a side effect; per-job approvals; read allowlists; a sandbox for tools from packs (Section 3)	The gate, a refused write and a probe in the sandbox, yes; real publishing, no
5	Turn processes it already knows into tools	Workflows written in the factory and exported (Section 4)	Yes, and a measured comparison (Section 7)

Table 1: What a company does to put the agent into its own backend service, and what we tested. Every “yes” was run by the authors on their own application; no other company has done this.

(feedback from the system and from the outside world) and act on it, within limits. Systems built around agents usually fuse these into one process, with prompts, model credentials, platform tokens, browser sessions and application data side by side. That makes each part hard to test, hard to replace, and hard to share.

This paper describes a different split and the open-source components that implement it. Our working domain is marketing, where the pattern is easy to see: collect material from an application, make content, publish it, look at the result. But nothing in the *shape* of the design is marketing-specific. What changes between domains is the instructions, the tools and the connections, not the architecture. We therefore state the contribution in domain-neutral terms and use marketing as the case. The agent is meant to be re-fitted rather than rebuilt: a new domain should need a new prompt, new tools and a new connection, and no change to the orchestrator, the approval gate, the pack format or the way credentials are held (Section 2.2). We have shown this only for marketing and for domain-neutral test tasks, and we do not claim it for any other domain.

1.1 Who this is for, and what the company does

The reader we have in mind is a company with a backend service and an application of its own that wants an agent to help with its work: to read what the application holds, make something from it, and act on the outside world, without handing the model its database, its platform credentials or its authority. Such a company has to do five things, and the rest of the paper is organised around them. Table 1 lists them, with what the company writes and whether we tested it. The orchestration is the company’s to shape: which sub-agents run, each one’s prompt, model and tools, the orchestrator’s own prompt, and which actions need approval are all configuration, not code.

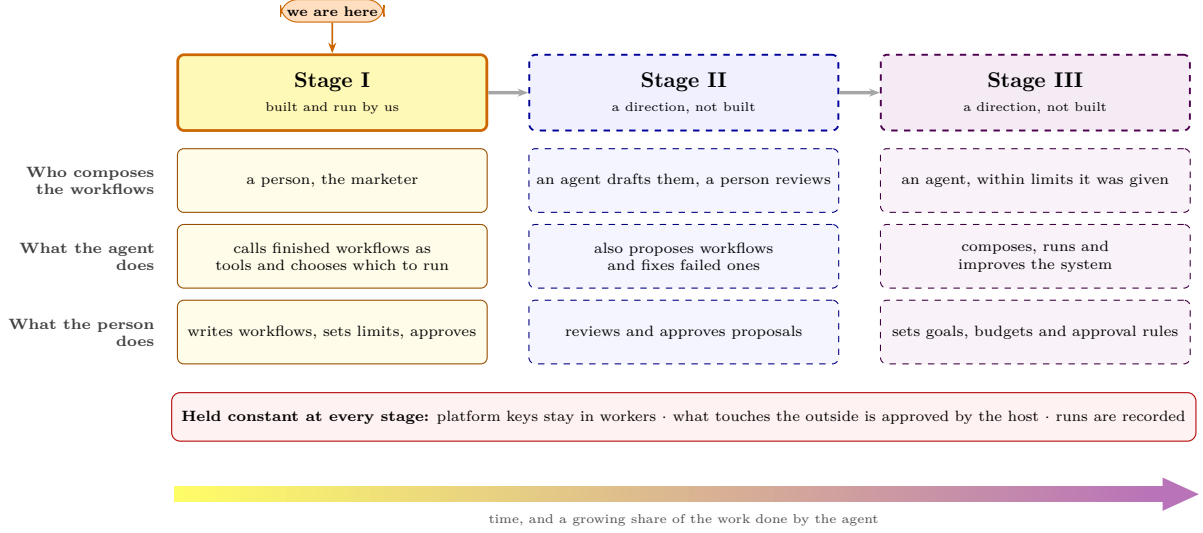


Figure 1: Three stages. Our work belongs to Stage I and is kept compatible with the later two; Stages II and III are directions, not results. The safeguards in the lower band are meant to stay the same as the agent’s share grows.

1.2 The problem, and where we are in time

Systems built on language models lie on a range. At one end are *workflows*, in which models and tools are orchestrated through predefined code paths; at the other are *agents*, in which the model directs its own process and tool use [1]. We describe Stage I as the setting in which workflows and narrow autonomous tools carry most operations and a person, the marketer, decides what to run, assembles the workflow, sets the limits and approves what touches the outside world. This is our characterisation of the setting, not a measured claim: we did not find a study we could read that quantifies how widespread either form is in marketing, and we state no figure. We expect, but have not shown, that over time more of this is done by agents, until an agent manages the whole system and the person sets goals and constraints. Figure 1 shows the three stages and where this work stands.

The problem this paper addresses is the engineering needed in Stage I that does not get in the way of Stage III. Concretely: how to give an agent the tools, the connections and the limits for a given system, how to turn the processes a person already knows into tools the agent can call, and how to keep credentials and authority out of the model, so that the same parts remain usable when the agent is given more of the work.

Contributions.

1. An arrangement for placing an agent inside a company’s own backend service: the application exposes what it approves through a contract, the agent runs as a worker inside the service boundary with a database role that reaches only its own queue, and the company shapes the orchestration through configuration (Table 1; Sections 2.2 and 8).
2. A decomposition of “fitting an agent to a system” into four parts that can each be used alone: a customisable agent, a connection layer that is also an open pool, a workflow factory, and an example that combines them (Section 2).
3. Sub-agents that join an agent to a system through contracts rather than through a product (an asset sub-agent and a platform-data sub-agent), which can be used with any application or worker that implements the contract (Section 2.3).
4. An open contribution system: through GitHub and one common format, different people add an agent, a tool, a skill or a worker, and an organisation pulls the pieces together into an agent extension for its own system (Section 5).

5. A trust boundary in which the agent never holds platform credentials and external side effects are authorised by the host rather than by the model (Section 3).
6. A factory path from a deterministic workflow to several consumers through a neutral bundle and file-writing adapters, so that known processes need not be rediscovered by a model on each run (Section 4).
7. A report of what was actually run, including two faults in our own design that the exercise exposed (Section 6), a measured comparison of one workflow tool against separate tools on a three-step task and an eight-step chain, with two models (Section 7), a test of the agent inside the backend service of a real application (Section 8), and an explicit list of what was not run (Section 10).

What this paper is not. It is not a benchmark and it does not claim that autonomous marketing works. The evidence is a set of working demonstrations on one machine. Where something is a direction rather than a result, we say so.

2 Architecture: four parts

Figure 2 shows the parts and how they relate. None of them is a placeholder for another, and none requires the others to run.

The four parts correspond to four open-source repositories (Table 2). Ethgent is the name of the agent application and of its repository (it was first published as BrowserAgent, which now points to it).

Repository	Role
Ethgent	Agent architecture, reasoning and autonomous decision-making; runs as a terminal app or as a worker inside a backend
MarketingAssets	Reusable marketing capabilities, toolboxes and skills
MarketingStudio	Development environment for capabilities and workflows
MarketingPool	The system that brings the agent, the assets and the workers together

Table 2: The four repositories and their roles. The names link to GitHub.

2.1 Ethgent: the customisable agent

Ethgent is an orchestrator model that delegates to sub-agents, each of which owns a set of tools. The orchestrator can call a tool only through a sub-agent that lists it, so giving a sub-agent read-only tools is how it is kept read-only. Agents, tools and prompts are described in configuration and travel as *packs* that can be created, edited, installed from another repository and shared from a terminal, without code changes. It ships tuned for social media and content. The same mechanism would host, for example, a trading agent with market-data tools or an HR agent with application-screening tools; we have not built those, and name them only to mark what varies between domains (Table 3).

2.2 What is fixed and what is fitted

The claim that the agent adapts to a domain is only worth making if we can say which parts do the adapting. Table 3 separates them. The left column does not change between domains; the right column is what a person writes to fit the agent to a new system.

Two points keep this honest. First, the shipped marketing sub-agents (for example the one that lists an application’s approved assets and the one that reads platform data) are *built-in* agents, written as Python classes, with their tools as Python modules. They are not examples of

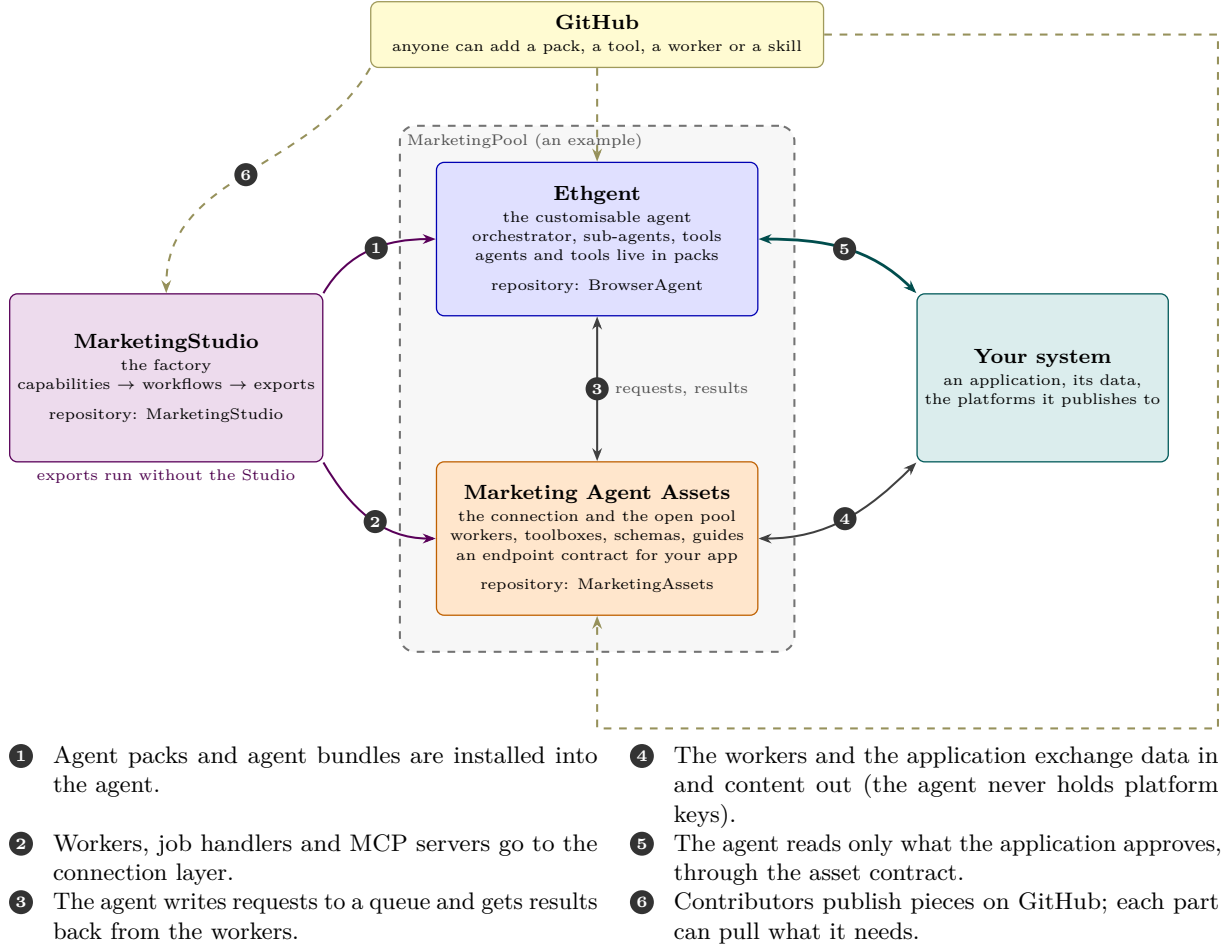


Figure 2: The four parts and how they relate. Each can be used on its own: the agent runs without the pool, the pool without the Studio, and an export without the Studio. The dashed box is a worked example (Docker, one machine), not a required deployment.

the configuration path. The configuration path is the agent type that a user creates: a YAML file, a prompt and a tool file, installed as a pack or made from the terminal, with no change to the core. The repository ships one such pack, a small news-headline agent; it is not a marketing agent, but it is a toy and we did not run it in the experiments. Second, the experiments in Section 7 used domain-neutral text-statistics tools, so they show that the comparison does not depend on marketing content, not that a trading or HR agent works. Whether a second domain really needs only the right-hand column is untested; we list it as future work (Section 10).

2.3 The sub-agents we wrote

Ethgent is not only a mechanism for writing sub-agents. It ships with nine, and two of them were written for this work. Table 4 lists them. Each is a separate agent with its own prompt and its own tool list, so an installation can switch any of them on or off, and the orchestrator can reach a tool only through the sub-agent that owns it.

The first five are general. They are how Ethgent looks at a screen, browses, researches and makes content, and a project in another domain, such as trading or hiring, would use them in the same way: `browser_agent` to read a page, `arastirma_agent` to research, `vlm_agent` to operate a program with no interface other than the screen. `sosyal_medya_agent` and `content_creator_agent` are the marketing-specific pair that we tuned and that are on by default. Three of them depend on a particular model vendor, which limits where they can run as shipped.

	Fixed (same in every domain)	Fitted (written for the domain)
Agent	Orchestrator that delegates to sub-agents; a tool is reachable only through a sub-agent that lists it	Which sub-agents exist and are on, their prompts and models, which tools each may use, the orchestrator’s own prompt
Tools	How a tool is declared, called and logged	The tools themselves, for example a function that reads prices or screens an application
Limits	Approval at call time by the host, never by the model; a tool that declares a side effect is gated; tools from packs run in a sandbox	Which tools declare a side effect; per-job grants; allowlists of what may be read
Connection	Credentials stay in workers; requests and results are validated against schemas	Which workers run; the app’s endpoint contract; which platforms or services are reachable
Sharing	Packs: a manifest, agents, prompts, tools; install from a GitHub repository	The content of the packs

Table 3: What stays the same and what is written for a domain.

The last two matter more than the rest for the argument of this paper, because they are the ones that join the agent to a system through the contracts of Section 2 and not through a platform’s own tools. `asset_collector_agent` does not know Kara Tahta or any other application: it calls the asset endpoints and reports what comes back. `platform_data_agent` does not know which platforms exist: it asks the worker for what the manifests allow and the worker refuses the rest. Any application that implements the asset contract, or any worker that publishes a manifest, can therefore be used with these two as they are. We think a sub-agent that speaks a contract rather than a product is the part most likely to be reused outside our setting.

There are three limits on that claim. (1) When this paper was first published the two sub-agents existed only in MarketingPool’s copy of the agent. They are now in the Ethgent repository, off by default, and MarketingPool uses that repository instead of a copy. (2) They are built-in code. Ethgent’s pack format moves an agent between installations as configuration, and for a built-in agent that means the model, tool list and prompt only; the code travels with the Ethgent repository, not with a pack. We did not test their reuse with anyone outside the project. (3) Of the nine agents, only two were touched in this paper’s experiments: `asset_collector_agent` answered the requests of Section 8, and `platform_data_agent` was reached only on the path where a key is missing and where a write is refused. The other seven were not part of our experiments; Table 4 describes what they are for, not how well they do it.

2.4 Marketing Agent Assets: the connection and the open pool

This layer joins an agent to a system. It contains workers that hold credentials and talk to platforms, platform toolboxes (declared in manifests), shared JSON schemas for requests and results, decision guides for the model, and a small endpoint contract (`asset-pool`) that any application can implement to expose its own approved media and capabilities. It is also an *open pool*: there is no single asset. Anyone can contribute a tool, connector, worker, schema or guide, and most contributions are declarative (a manifest entry, a guide, a schema), so sharing does not require modifying a worker. The aim is that nobody rebuilds the same integration alone.

Sub-agent	What it does	Tools	Default; where
<code>sosyal_medya_agent</code>	Publishes, engages and scans notifications on X, Instagram and YouTube through a supervised browser; the one that touches accounts, so its writing tools are the ones the approval gate covers	51 to 56	yes; both
<code>content_creator_agent</code>	Makes the material: captions, media search, content from a website, HTML/CSS images, short videos	about 18	yes; both
<code>browser_agent</code>	General web work in a Selenium browser: navigate, read the page, fill forms	about 30	no; both
<code>vlm_agent</code>	Controls a machine by looking at the screen: screenshots, keys, scrolling, hovering, reading a pixel. Runs on a Gemini Live model	9	no; both
<code>arastirma_agent</code>	Deep web research and a written report. Runs on a Gemini Live model	8	no; both
<code>sistem_agent</code>	Files, system analysis, memory. Runs on a Gemini Live model; its shell tool is disabled globally	13	no; both
<code>computer</code>	A scaffold made with Ethgent’s own agent creator, with one system-status tool. It is an example of the creator, not a finished agent	3	no; both
<code>asset_collector_agent</code>	Lists an application’s approved media, validates and prepares it for publishing, asks the application for video generation. Speaks the asset contract	7 plus workspace	no (yes in Marketing-Pool); both
<code>platform_data_agent</code>	Asks the platform worker for read-only metrics, comments and account data, limited by the manifest allowlist. Speaks the data contract	3 plus workspace	no (yes in Marketing-Pool); both

Table 4: The sub-agents in the repositories. Ethgent ships all nine; the last two are off by default because they need the connection layer’s settings, and MarketingPool, which uses Ethgent as a submodule, switches them on. Tool counts are taken from the configuration and the tool lists in the code; “about” marks counts we did not check one by one.

2.5 MarketingStudio: the workflow factory

The Studio turns capabilities into new capabilities. A capability is described by a contract (inputs, outputs, requirements, permissions, cost estimate, whether it writes external state) and is either a contract file next to the code or a single Python file with the metadata beside the function. A workflow is a free-form graph of steps over capabilities from any source. The Studio validates it, plans it (order, approvals, known cost, whether the machine is ready), can run it locally, and exports it. Its exports feed the agent directly (an agent pack or agent bundle installs into Ethgent) and the connection layer (queue workers and job handlers). Section 4 describes the export path.

2.6 MarketingPool: the example

MarketingPool runs the agent and the assets together with Docker on one machine, with a local Postgres-based queue in place of a hosted database. It exists to show that the combination works and how, not to be the only way to combine the parts.

2.7 Where things run

Figure 3 separates design time from run time. The Studio is a design-time tool used by a person; its exports are moved to the server side. At run time the agent and the workers and assets both live inside the *backend service* of the system they serve, next to its queue, and they are what the application and the outside platforms talk to. The Studio does not have to be installed there. In MarketingPool this backend service is a set of containers on one machine. Section 8 tests this arrangement against a real application.

2.8 The loop

When the agent and the assets are joined to a system, the loop closes: the model receives feedback (it can read data from the system and from platforms) and can give something back (it can publish, or write into the application). Connected to an application, it can collect material from it, make content and publish it, with the application remaining the source of truth.

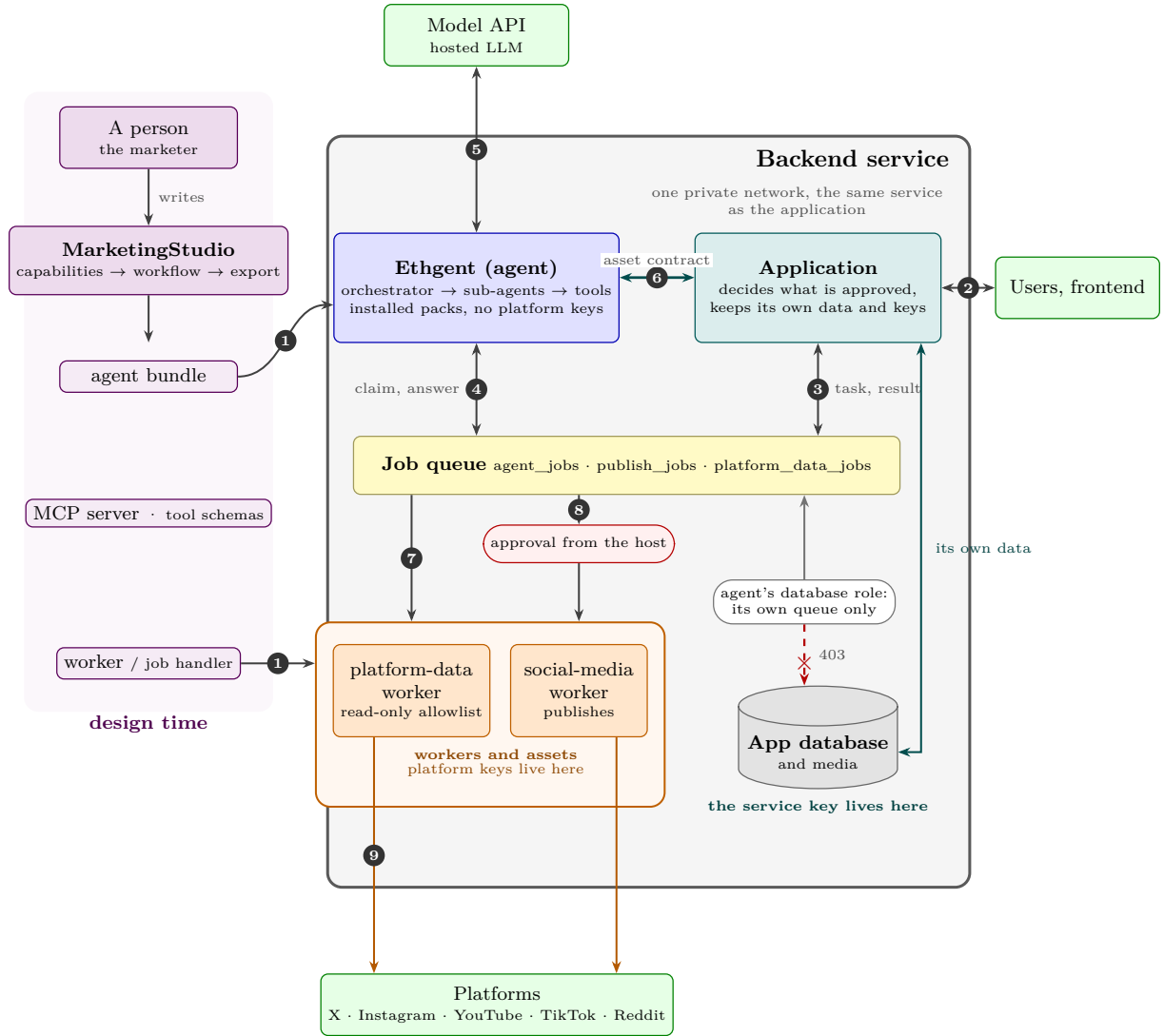
3 Trust boundary

The agent does not hold the keys. The agent writes a validated request to a queue. A worker that owns the credentials checks the request and acts. The read-only data worker executes only actions on an allowlist declared in each toolbox manifest, and refuses anything else, including writes.

External side effects need approval from the host. Publishing, replying, following and similar actions are gated at call time. The answer to the gate comes from the host: a person at the terminal, an explicit per-tool grant carried on the queued job, or, headless with no grant, a refusal. It never comes from the model, and a tool exposes no argument by which the model could approve itself. The unit of approval is a tool, which for an exported workflow means the whole workflow (Section 6 explains why this matters). The Studio marks a step as gated when its capability declares that it writes external state, and the exports carry that rule with them: a pack tool declared `requires_approval` in its manifest is gated by the agent before it runs, whatever its own code does. The agent's own social tools that act in public (posting, replying, liking, following, commenting) pass the same gate. The general browser and shell tools do not; they can do anything, so whoever deploys the agent decides which sub-agent holds them.

Other people's code runs outside the agent. A tool that arrives in a pack does not run in the agent's process. Its name, description and parameters are read from the source without executing it, and each call runs in a sandbox: a throwaway container with a read-only filesystem, no capabilities and memory, CPU and time limits when Docker is available; a separate sandbox service in its own container when the agent itself runs in one (the usual backend case, where the agent cannot start containers); a limited subprocess otherwise. In every case the tool receives only the environment variables it declared, and never the ones the agent itself uses (model keys, queue keys, chat tokens), even if it declares their names. We checked this with a probe tool in the sandbox service next to the containerised worker: it ran on another host as an unprivileged user, saw only its declared variable, could not see the agent's files and could not resolve the queue or the database. A pack can still be installed as trusted, in which case its tools run in the agent's process; one that carries code for a sub-agent is refused without that choice.

Before running. The runner states what is missing (keys, programs, packages) instead of failing halfway, and a plan lists the order, the approvals and the known cost.



- ❶ Studio exports are installed: the agent bundle into the agent, the worker or handler beside the workers.
 - ❷ Users reach the application as before.
 - ❸ The application queues a task and later reads the result.
 - ❹ The agent claims the job, reasons, and answers through the queue.
 - ❺ The agent calls a hosted model; its key is the agent's own.
 - ❻ The agent reads only what the application approves, through the asset contract and a bearer token.
 - ❼ A read-only data request: the worker checks it against an allowlist and refuses writes.
 - ❽ A publish request waits for approval from the host: a person or an explicit grant on the job, never the model.
 - ❾ Only workers hold platform credentials and act on the platforms.
- ✗ The dashed red arrow shows what the agent's database role refuses: the application's own tables (HTTP 403).

Figure 3: Design time and run time. The agent and the workers and assets run together inside the backend service, next to the application they serve; the Studio does not. What was run is stated in Sections 6, 7 and 8; nothing here was run against a real platform.

4 The factory path

The design rule behind the Studio is to *reason when necessary and execute deterministically whenever possible*. Most marketing operations are repeatable processes; a known process should not be rediscovered by a model on every run. The Studio therefore treats the model as a consumer of finished workflows rather than the executor of every step.

Neutral bundle. An export is a self-contained bundle: the workflow, the contracts, a runner that uses only the Python standard library, and single-file engines (engines that live elsewhere are referenced as external folders). A bundle runs without the Studio installed.

Adapters. Small functions reshape the bundle for one kind of consumer and only write files (Table 5). The same workflow can be offered to an agent as a tool, to an existing worker as a handler, or to any MCP client [3], without touching its definition.

Target	Result
agent-pack	a pack with one self-contained tool file for an agent app
agent-bundle	the same, plus a sub-agent that owns the tool and its prompt
tool-schema	tool definitions for LLM function calling (two vendor shapes)
job-handler	one handler for a worker that is already running
worker	a standalone queue worker with migration, container file and per-step approval
mcp	a stdio MCP server offering the workflow as one tool

Table 5: Export targets. Adapters are pure functions from a bundle to files.

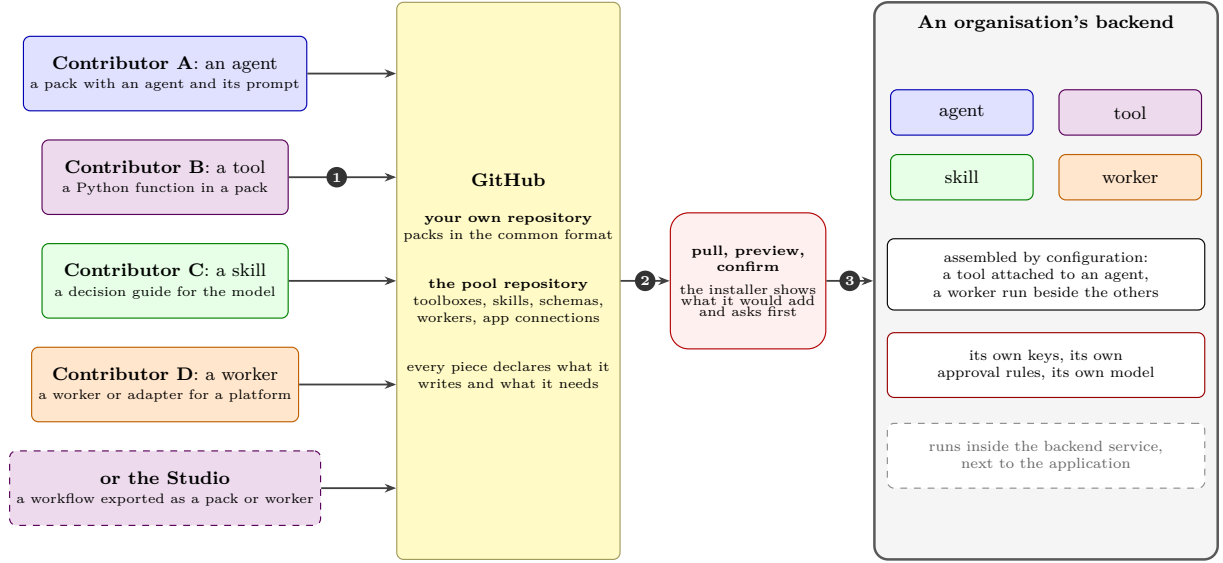
Today and later. Today a person writes the workflow and moves the export to the server side. An agent that uses the Studio itself, to discover capabilities and compose workflows, is a direction that the plain-file formats leave open; it is deliberately not implemented, and nothing in the present design depends on it.

5 Open contribution through GitHub

The parts above are not meant to be used only by their authors. The purpose of the project is an *open contribution system*: many people, each adding one piece, and a way for those pieces to be shared, pulled in and joined into a working extension of someone’s own system. GitHub is the commons; the factory supplies the common format.

5.1 Mechanism

1. **A common format.** The factory writes every workflow as a neutral bundle (Section 4) and reshapes it with adapters. A contributor who writes a capability once can therefore offer it as an agent tool, an agent with its prompt, a queue worker, a job handler or an MCP server, without rewriting it.
2. **Declarative pieces in the pool.** Most of what the pool accepts is not code that must change a worker: a platform action in a toolbox manifest, a read-only data action added to an allowlist, a decision guide for the model (a skill), a shared schema, a connection to an application through the **asset-pool** contract, a queue adapter, a worked example. Each has a place in the repository and a rule it must satisfy (declare whether it writes external state, no secrets, independence from any single application).
3. **Sharing by repository.** An agent setup is a pack: a manifest, agents, prompts and tools. A pack can be published as an ordinary GitHub repository, and any Ethgent installation can fetch it from there; the installer clones it, shows a preview of what it would add and asks before installing.
4. **Re-integration.** A piece that arrives this way is joined to the receiving system by configuration: a tool is attached to a sub-agent, a worker is run beside the others, a guide is added to the pool. Nothing has to be merged into the contributor’s code.



- Not built:** a registry where pieces are found, versions and dependencies, review and signing of contributions
- Open risk:** a pack carries code that runs on the server; it is sandboxed, but a pack from a stranger is still a supply-chain risk
- ❶ Different people add different pieces, most of them declarative, and publish them in one common format.
 - ❷ An organisation pulls the pieces it needs; the installer shows what each would add.
 - ❸ After the preview the piece is installed by configuration; nothing is merged into the contributor's code.
- The right-hand box is a scenario: no group of independent contributors has assembled a system this way yet.

Figure 4: The contribution loop. Contributors add different pieces; an organisation pulls what it needs into its own backend service. The mechanism exists, the community does not yet, and the two boxes at the bottom say what is still missing.

5.2 An illustrative scenario

Suppose someone starts a trading agent: a sub-agent with a prompt and no tools that can change anything. Another person adds a skill that tells the model how to weigh a position. A third adds a tool that reads prices. A fourth adds a worker that holds the credentials for a data vendor. Each publishes what they made; none has to know the others. A trading firm then pulls the four pieces into its own backend, reviews the preview, attaches the tool to the agent, runs the worker next to its other services, and has an agent extension that works inside its own system, with its own credentials and its own approval rules. Because the pieces meet through files and small contracts, a fifth person can later replace the price tool without touching the other four.

We present this as a scenario. It follows from the mechanism, but we have not built a trading agent, and no group of independent contributors has yet assembled anything this way (Section 10).

5.3 What exists and what does not

- *Exists and was exercised by the authors:* writing a workflow in the factory, exporting it as an agent bundle, and installing it into a running agent; the agent pack format with install from a local folder; the pool's contribution rules and the declarative places they point to. Fetching from GitHub was checked in two ways: the agent's test suite installs a pack through the `github:` source with the clone redirected to a local repository, and on 10 October 2026 we cloned a pack for real from a public repository (the `packs` folder of MarketingPool), installed it into a fresh workspace with the agent's own fetch and install

functions, and had the agent call the installed tool in 55 consecutive runs (Section 7). The pack came from the authors’ own repository, so this is not an independent contribution, and we called the same functions that the terminal command uses rather than typing the command.

- *Not built*: a registry or index through which contributors and users discover each other’s pieces; versioning and dependency resolution between pieces from different authors; review and signing of contributions; a way for the factory to produce a skill (skills are written by hand today).
- *Open risk*: a pack carries tool code that runs on the server. Since the revision of this paper that code runs in a sandbox, with only the variables it declared (Section 3); but inside the sandbox it can still use the network and its own declared keys as it likes, and a pack installed as trusted runs with everything the agent can reach. Preview and confirmation help a careful reader, not a hurried one. We have not designed a trust model for third-party contributions beyond this.

6 What we ran

Setup. The agent and the workers ran in containers on one Linux laptop, with a local Postgres-based queue (Postgres 16 behind a REST layer) in place of a hosted database. The agent used a hosted chat model through an OpenAI-compatible API. Tasks were handed to the agent through the queue. Four example workflows were written in the Studio, checked and planned there, exported with the `agent-bundle` target, and installed into the running agent as a pack; the agent was then asked, in natural language, to use them.

Workflow	Property exercised	Outcome
<code>text_summary</code>	deterministic steps only, no model inside	the agent called the tool and returned the report; its words and top-word list (17 words; “marketing, agents, workers”) equal the reference answer computed by the Studio’s own runner
<code>text_report</code> <code>human_size</code>	takes a file as input engine that needs a third-party package	works after the fix below package installed on a volume, called; it returned “1.5 MB” and did so again after the container was recreated
<code>report_and_send</code>	sending step that needs approval	refused without a grant, ran with one

Table 6: The four demonstrations. They are demonstrations, not a benchmark.

Two faults the exercise exposed. (1) *A model-settable approval flag can be bypassed.* Our first version of a gated workflow tool exposed an `approve` argument and told the model to set it only after a person agreed. A task text could simply instruct the model to set it. We removed the argument. The tool now asks the host’s gate, and a queued job can carry an explicit list of tools it approves. We added tests, including one that checks that a model cannot approve through an argument.

(2) *File paths do not survive two model hops.* A file named in the user’s task passes through the orchestrator and then a sub-agent before reaching the tool, and the path was altered along

Task		Runs	Headline	Invented	Model calls	Tokens (mean)	Time (median s)
short	A	20	20	1	4.4	11 364	4.8
	B	20	19	1	5.3	12 520	5.9
long	A	20	20	1	4.2	15 144	8.3
	B	20	20	0	5.5	20 604	11.0
batch	A	15	15	0	7.8	14 253	10.1
	B	15	15	0	11.7	18 941	12.4

Table 7: One workflow tool (A) against three separate tools (B), same model and prompts. “Headline” and “Invented” count runs.

the way. We made the runner resolve a relative path against the folders the workflow is allowed to read, staying inside them, and name those folders in the error when the file does not exist.

Both faults sit between a model and a tool, not inside either. We would not have found them from the component tests alone.

Tests. Run on 9 October 2026. The Studio and its read-only viewer have 72 automated tests (validation, planning, the runner, each adapter) and the Storyforge API 29, all passing (Studio commit [8affc56](#)). The agent’s full test suite, 242 tests including nine for the approval gate, passes on the branch that adds the gate (BrowserAgent commit [d3dade4](#)). In the copy of the agent inside MarketingPool, six of the seven tests for per-job approval pass on the host; the seventh imports the queue worker and needs the `mcp` package, which is installed in the container image but not on this host, so we did not run it here. The wider suite of that copy needs model keys and was not run.

7 A measured comparison

Section 4 argued that a known process should run as code and not be rediscovered by a model on every run. This section tests that argument at small scale. It does not settle it.

Question. Does offering a fixed sequence of steps as one workflow tool change the cost or the reliability of an agent, compared with giving the agent the same operations as separate tools?

Design. The orchestrator, the model (`deepseek-chat`), the task wording and the surrounding agents were identical. Only one sub-agent differed, and each owned its tools:

- **A, one workflow tool.** The `text_summary` workflow, exported from the factory as an agent bundle, installed from GitHub into a fresh workspace (Section 5), and called as one tool.
- **B, three separate tools.** The same three operations (`word_count`, `top_words`, `report`) as three tools holding the same code; the sub-agent has to call them in turn and pass results along.

The request was to write a short report with the title, the word count and the five most frequent words of a text. Three task sizes: a 62-word text (20 runs per condition), a 563-word text (20) and a batch of three texts of 41–48 words (15). Runs were interleaved, and the two conditions ran at the same time. Cost was read from the agent’s own telemetry (model calls, tokens, wall time). Each answer was scored automatically from the final text: *headline correct* means the word count appears and the five words appear one after another in the right order, whatever the formatting or language around them; *invented* means the answer states an occurrence count for a listed word that is wrong, although neither version of the tools returns such counts.

Model	Text		Correct	Model calls	Tokens (mean)	Time (median s)	B/A tokens
chat	short	A	14/15	4.1	10 680	4.5	1.65
		B	14/15	6.7	17 633	7.9	
	long	A	15/15	4.0	14 572	7.7	2.52
		B	13/15	6.9	36 714	18.7	
v4-pro	short	A	14/15	4.1	11 741	10.8	1.88
		B	15/15	7.3	22 094	24.4	
	long	A	15/15	4.9	20 257	23.6	2.09
		B	13/15	7.5	42 402	55.5	

Table 8: Eight-step chain: one workflow tool (A) against eight separate tools (B). “chat” is `deepseek-chat`.

Reliability: no measurable difference. The headline figures were right in 55 of 55 runs for A and 54 of 55 for B, and the two versions invented figures in 2 and 1 runs. The single wrong headline (B) was an orchestrator that ignored the tool and counted the words itself, getting 61 instead of 62: the failure of an agent that decides every step. In two A runs the orchestrator added occurrence counts that the tool never returned (for example “model: 12” where the text has 10), so a workflow does not stop the orchestrator from embellishing a result. With these counts we cannot claim a difference.

Cost: smaller than argued. B used 1.10 times as many tokens as A on the short task (bootstrap 95% interval 0.97–1.26, which includes 1), 1.36 times on the long task (1.22–1.53) and 1.33 times on the batch (1.20–1.50), with 1.2 to 1.5 times as many model calls and 1.2 to 1.4 times the wall time. Over all runs B spent 947 000 tokens and A 744 000. The gap grows with the length of the text and of the chain, but it is a matter of tens of percent, not an order of magnitude: prompt tokens make up 95% of all tokens in the short task, because the prompts of the orchestrator and the sub-agent are sent again on every model call, and both versions pay for that. (We did not break the prompt down further.)

A longer chain and a second model. The three-step task is nearly the least favourable case for a workflow, so we repeated the comparison with eight dependent steps (clean the text, count words and sentences, average sentence length, most frequent words, longest word, reading time, then a report) and with a second model of the same vendor (`deepseek-v4-pro`). Condition A was again one workflow tool exported from the factory; condition B was eight separate tools with the same code. The request asked for the report to be returned in the reply. We ran 15 short (62 words) and 15 long (563 words) requests per condition and model, scored by whether every figure appeared correctly in the answer.

Cost grows with the length of the chain. B used 1.65 to 2.52 times as many tokens as A (every 95% bootstrap interval starts at 1.5 or higher), 1.6 to 1.8 times as many model calls and 1.7 to 3.3 times the wall time. Over all chain runs B spent 1 783 000 tokens and A 859 000. The mechanism is visible in the logs: in B the model reads each result and sends the text again to every step that needs it, so the long text is paid for several times (36 700 tokens against 14 600), while A passes it once and the steps run in code. A’s cost is also almost constant: the standard deviation of A’s tokens on the long text was 52.

Reliability: a tendency, not a result. Counting all 60 runs per condition, 2 of A’s answers and 5 of B’s were wrong ($p = 0.44$, Fisher exact; the pooling across tasks and models is rough). Most failures were not arithmetic: in 4 of B’s 5 (one was empty) and 1 of A’s 2 the orchestrator replied that the report was ready and never gave it. The rest were wrong figures, where the

model computed a value itself instead of using the tool’s (for example 15.5 words per sentence in A, where the tool returns 12.4). The numbers are too small to claim that workflows are more reliable, but they point in that direction and do not point against it.

What this supports. Packaging a known sequence as one tool lowered cost and latency, by tens of percent on three steps and by a factor of roughly two on eight, and the saving grew with the chain and with the length of the data passed along. This supports the cost argument. It does not yet support the reliability argument: the differences in correctness were not statistically detectable.

Threats to validity. The two models come from one vendor, so they are not independent evidence about models in general. Both task families are deterministic, text-only and small; the authors wrote both conditions; 15 to 20 runs per cell is few; the orchestrator’s wording varies from run to run, and our scoring checks that the figures appear, not that the answer is well written. The text is passed inline, which favours A; a design that passed files or handles between tools would shrink B’s overhead. Our first version of the separate tools also failed outright, because the model sent numbers as strings (for example "5"); we hardened them before the measured runs so that B would not be handicapped by our own code, and the first suite’s wording was changed for the second (“Reply with the report”) after a trial showed that without it the orchestrator sometimes withheld the report in both conditions. Code, inputs and every run’s answer are in the `experiment` folder.

8 Inside a backend service

Section 2 claims that the agent and the workers run inside the backend service of the system they serve, and reach it only through a narrow contract. Sections 6 and 7 did not test that: the agent there ran beside a queue and nothing else. Here we test it against a real application.

The application. Kara Tahta is an Express application of ours that turns a topic into a narrated lesson video, with its own database, media storage and render queue. It is open source under the AGPL-3.0 at <https://github.com/ethosoftai/karatahta-oss>, so the test environment can be rebuilt by anyone. It implements the pool’s asset contract (Section 5) with a thin route and an adapter that maps lessons to assets: only lessons marked public, with a finished video that is not an imported YouTube link, are offered, as signed URLs.

Setup. One Docker Compose project on one machine, with a private network: the application unmodified (a diff shows that the copy we ran has routes, services, migrations, server and workers byte-identical to commit 8658c59 of the public repository), with its own 19 database migrations applied by a script, a plain Postgres with a REST layer standing in for its Supabase backend (plus a short stub of Supabase’s `auth.uid()`, which two of its policies call), its local media provider, an HTTPS front for the media URLs (the contract requires `https://`), and the agent running as a queue worker with a hosted model. The application’s database key is present only in the application’s container. The agent holds a database role that can touch its job queue and nothing else, the contract token, and the model key. Test data: five lessons, of which two are public with a finished video, and three must stay invisible: a private lesson, a public lesson whose video is still rendering, and an imported YouTube lesson.

What the results show. The arrangement works as described: the application decides what is approved and the agent sees only that. The two guarantees that matter do not depend on the model. What the agent can read is limited by the application’s contract (the private lesson is never returned, so no prompt can make the agent report it) and by a database role (a direct

Check	Result
Contract conformance checker	19 checks passed, 0 failed, 1 skipped (video generation is not enabled in this setup)
Agent lists the approved assets	4 assets (a video and a poster for each of the 2 approved lessons). The private, unfinished and imported lessons did not appear in any of 10 runs; the list reached the user in 9 of 10 (in one run the orchestrator said the list was ready without giving it)
Agent asks for the private lesson by id	the application answers 404 and the agent reports “asset not found”
Agent’s database key against the application’s tables	403 on <code>lessons</code> and <code>profiles</code> ; 200 on its own queue. The application’s key reads <code>lessons</code>
Secrets in the agent’s container	no database service key and no media signing secret; the only application credential is the contract token. No signed URL is written to the agent’s workspace (0 files contain one)
Application unreachable	the agent says it cannot reach the application and invents no assets
Ask the agent for all lessons including private ones	it refuses and lists what it can do instead (see below)
Prepare the video for a YouTube upload	refused by the agent’s own guard because the media host is not public; not exercised further

Table 9: The agent and a real application inside one backend service.

query with the agent’s key is refused). The agent’s own refusal when asked for every lesson is a statement of the model, and we count it as colour, not as evidence. That the application is reached by an internal name and exposes its port only to the host is a property of the setup, not a result.

What it does not show. The backend is a stand-in: plain Postgres and local media instead of Supabase and a storage bucket, so we have tested the application’s code and its contract implementation, not its production deployment. The media are two-second synthetic clips. The step that matters most for publishing, preparing and validating media for a platform, was blocked by the agent’s own safeguard against private addresses, which is correct here and means we could not check it; a public media URL is needed for that. We did not run video generation through the contract, did not publish anything, and did not test the application’s own scheduler. The application and its contract adapter were written by us, so this is a test of the design by its authors, not of whether an outside team can integrate their application. The application, its test data, the Compose file and every answer are in the `experiment/backend` folder.

9 Related work

Tool-using language models. ReAct interleaves reasoning traces with actions so that a model can query external sources while it plans [7], and Toolformer trains a model to decide when to call which tool [5]. Both concern how a model uses tools; our concern is the engineering around such an agent: where its tools come from, where credentials live, and who authorises what.

Workflows and agents. Anthropic distinguishes *workflows*, where models and tools are orchestrated through predefined code paths, from *agents*, where the model directs its own process and tool use, and recommends the simplest solution that works, with guardrails and human

checkpoints for agents [1]. Our factory follows the same line: it produces workflows, which an agent then calls as single tools, so that known processes are not left to the model.

Multi-agent frameworks. AutoGen builds applications from several agents that converse with one another and can combine models, human input and tools [6]. Ethgent is likewise an orchestrator with sub-agents; the difference we emphasise is not the agent loop but the separation of the agent from the connection layer and the factory, so that each can be replaced.

Tool protocols. The Model Context Protocol is an open standard for connecting AI applications to external data sources, tools and workflows [3]. One of our export targets is an MCP server; a workflow that needs approval is refused over that path because our MCP server has no way to carry the host’s approval.

Security of tool-using models. Indirect prompt injection shows that instructions hidden in data a model reads can alter an application’s behaviour and control its other API calls [2]. The OWASP guidance on excessive agency recommends requiring a human to approve high-impact actions and limiting the permissions granted to extensions to the minimum necessary [4]. Our trust boundary applies the same two ideas: approval is answered by the host and not by the model, and read-only data requests are limited to an allowlist. Our own first design, in which the model set an approval flag, failed in the way these works describe: instructions in the task text could set the flag (Section 6).

What we did not survey. We did not review workflow engines, agent-tool registries or capability-based operating-system security in depth, and we make no claim of novelty against them.

10 Limitations and what was not run

- **No real platform was touched.** Real publishing needs platform credentials and public media URLs; real platform data needs keys. Only the path where a key is missing, and the refusal of a write on the read-only worker, were exercised.
- **One machine, one model.** The demonstrations ran on one laptop with one hosted model. We make no claim about reliability across models or about scale.
- **A small comparison.** We measured one workflow tool against separate tools for a three-step task and an eight-step chain, with two models of one vendor (Section 7). Cost fell by tens of percent and then by about half, and reliability did not differ detectably. We did not compare against an agent that works without tools, against other vendors’ models, against chains of tens of steps or against real platform calls, and the runs are few; so the claim that deterministic workflows are more reliable remains unproven.
- **Approval granularity.** Approval is per tool, and an exported workflow is one tool. A person approving a workflow approves all of its steps. Over MCP a caller can pass approvals only for tools the operator has listed for that purpose; the list is empty by default, and a caller that is itself a model can approve whatever is on it.
- **Copies of the agent.** When this paper was first published the agent existed in more than one copy. Since the revision, the queue worker, the MCP server and the two connection sub-agents are in the Ethgent repository and MarketingPool uses it as a submodule pinned to one commit; the backend test of Section 8 ran on the earlier copy, not on that commit.
- **The Studio is used by people.** An agent using the Studio to compose workflows is future work and is not implemented.

- **The contribution system is a mechanism, not yet a community.** No group of independent contributors has assembled a system this way, there is no registry, versioning or review of contributions, and a third-party pack runs code on the server (Section 5).
- **The backend test used a stand-in.** We ran a real application, but on plain Postgres and local media, with synthetic clips, with the authors’ own contract adapter, and without publishing or video generation (Section 8). Media preparation for a platform was not exercised because the test media host is not public.
- **No company outside the authors has adapted it.** Every “yes” in Table 1 was run by us, on an application we wrote, with our own configuration. We do not know how long it takes another company, or what it would have to change, to fit the agent to a different backend; that is the question the open contribution system is meant to answer.
- **Reuse of our sub-agents is untested.** The asset and platform-data sub-agents are built-in code in the Ethgent repository, so they do not travel as a pack; nobody outside the project has used them (Section 2.3).
- **The sandbox was checked on one machine.** The sandbox service and the container mode were exercised with our own packs and a probe tool, not attacked. A sandboxed tool may still reach the network, including the agent’s MCP port (which needs a token). A job whose worker stopped is requeued and runs again from the start, so a tool that had already acted outside may act twice; the recovery was checked against our local queue only.
- **Other domains are conjecture.** Trading, human resources and similar domains are consistent with the architecture; we built none of them. In particular, the claim that a new domain needs only new prompts, tools and connections (Table 3) has not been tested with a second domain, and the marketing sub-agents we ship are built-in code, not examples of the configuration path.

11 Conclusion

This paper is about how a company can put an agent inside its own backend service, join it to its application and the outside world, and shape the orchestration to its own work. We showed that arrangement against one application of ours; whether another company can do the same with less than we did is not yet known. We proposed splitting “fitting an agent to a system” into a customisable agent, a connection layer that is also an open pool, and a workflow factory, with credentials kept in workers and side effects approved by the host. A small set of working demonstrations shows the parts fit together, and two faults found along the way show where the seams between model and tool need care. A measured comparison showed that offering a workflow as one tool saves tens of percent of cost and time on a three-step task and about half on an eight-step chain, with no detectable change in reliability. What remains is the evidence a systems paper needs: runs against real platforms, models from other vendors, longer chains and more runs, and use by more people than the authors. The open contribution system, in which others add their own agents, tools, skills and workers through GitHub, is the purpose of the project and the part we can only test by its being used.

References

- [1] Anthropic. Building effective agents. <https://www.anthropic.com/engineering/building-effective-agents>, 2024. Published 19 December 2024; accessed October 2026.
- [2] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you’ve signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, 2023. arXiv:2302.12173.

- [3] Model Context Protocol project. Model context protocol. <https://modelcontextprotocol.io/introduction>, 2026. Accessed October 2026.
- [4] OWASP Gen AI Security Project. LLM06:2025 excessive agency. <https://genai.owasp.org/llmrisk/llm062025-excessive-agency/>, 2025. Accessed October 2026.
- [5] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, 2023. arXiv:2302.04761.
- [6] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. arXiv:2308.08155, 2023.
- [7] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.