



OpenText™ Fortify Agentic Analyzer

User Guide

Version: 26.4.0

Table of Contents

1	Introduction to Fortify Agentic Analyzer	1
1.1	Changes since 26.4-preview5	1
1.2	What Fortify Agentic Analyzer does	2
1.3	Fortify Agentic Analyzer and Fortify AI-powered SAST	3
1.4	Where to start	5
1.5	Conventions in this guide	5
2	Concepts and architecture	6
2.1	Hybrid SAST as a complementary approach	6
2.2	General architecture	8
2.3	Communication architecture	9
2.4	Coding agent options	10
2.5	The analysis process	12
2.6	SARIF output and the Fortify taxonomy	13
3	Installation and setup	15
3.1	The local execution model	15
3.2	Prerequisite: Fortify CLI (fcli)	16
3.3	Installers	17
3.4	Install Fortify Agentic Analyzer	17
3.4.1	Interactive installation	19
3.4.2	Unattended installation	20
3.4.3	The PATH entry	22
3.4.4	Upgrade and uninstall	23
3.5	Fortify entitlement (the license gate)	24
3.6	Set up the coding agent	25
3.7	LLM access for the Pi agent (bring your own LLM)	26
3.7.1	Credentials from the environment	26
3.7.2	Store credentials with -pillogin	27
3.7.3	Subscription policies: why only GitHub Copilot	28
3.7.4	Choose the provider and model	29
3.8	Verify your setup	31
4	Running a scan	32
4.1	The recommended workflow	32

4.2	Run a basic scan	32
4.3	Scope a scan	33
4.4	Complementary scanning	33
4.5	Output files	34
4.5.1	Choose formats	35
4.5.2	Name the artifacts	35
4.5.3	Keep the SARIF	36
4.5.4	What the SARIF contains	36
4.5.5	Markdown reports	37
4.5.6	Convert existing results	39
4.5.7	Message format	39
4.5.8	Restricted character set (<code>--latin1</code>)	40
4.6	Scan state and resume	41
4.7	Keep findings stable across scans	43
4.8	Change-driven scans: incremental and differential	45
4.9	Manage cost during a scan	46
5	Command reference	47
5.1	Commands (verbs)	47
5.2	<code>-scan</code> options	48
5.3	<code>-convert</code> options	52
5.4	Selecting and overriding configuration	53
5.5	<code>-selftest</code> options	54
5.6	<code>-license</code> options	54
5.7	<code>-pilogin</code> options	54
5.8	Where scan state is saved	55
6	Configuration: the policy file	56
6.1	Global settings	56
6.2	File enumeration	58
6.3	Per-stage model selection and effort	59
6.4	Plan stage	59
6.5	Analysis stage	60
6.6	Validation stage	61
6.7	Example: a budget-capped complementary scan	62
7	Extending Fortify Agentic Analyzer for your environment	63
7.1	Custom models and providers (<code>pi_models.json</code>)	64
7.1.1	Point a built-in provider at your endpoint	65
7.1.2	A model entry is a behavioral contract, not a label	66
7.1.3	Is the Pi model catalog exhaustive?	67
7.2	Extensions (<code>pi_extensions/</code>)	67
7.2.1	What an extension looks like	68
7.2.2	Rename a model on the way out	69

7.2.3	Develop and test an extension	70
7.2.4	Limits and responsibilities	72
7.3	Diagnose a custom setup	73
8	Consuming results	75
8.1	How Fortify Agentic Analyzer structures its SARIF	75
8.1.1	What this means in practice	76
8.1.2	The trade-off	76
8.1.3	Which code was scanned	77
8.2	Fortify on Demand (FoD)	77
8.2.1	Full detail in the FoD issue view	78
8.3	Fortify Software Security Center (SSC)	78
9	Agentic safety	79
9.1	The safety principle: read-only and offline	79
9.2	How it is enforced: an allow list of tools	79
9.2.1	Pi (bundled, default): an in-process capability boundary	80
9.2.2	Claude Code (external, opt-in): deny-by-default controls	80
9.3	Scope and limits of the guarantee	81
9.4	Do you need to run Fortify Agentic Analyzer in a sandbox?	81
10	Performance and results	83
10.1	Cost and speed	83
10.2	Determinism and repeatability	86
10.3	Completeness	87
10.4	Detection results on real CVEs	87
11	Troubleshooting	91
11.1	Locate the log files	91
11.2	Report issues and request enhancements	92

1 Introduction to Fortify Agentic Analyzer

OpenText™ Fortify Agentic Analyzer (FAA) is an AI-powered static application security testing (SAST) capability for the OpenText Fortify portfolio. It drives an autonomous coding agent over a project's source tree to uncover security vulnerabilities that lie beyond the reach of deterministic analysis. It complements, rather than replaces, the broad suite of purpose-built analyzers in Fortify SAST, and it emits its results in the standard Static Analysis Results Interchange Format (SARIF 2.1.0) for consumption by Fortify on Demand (FoD), Fortify Software Security Center (SSC), and other SARIF-aware tools.

FAA is a new *scan type*: run infrequently and risk-based, at a real (and significant) per-scan cost, alongside the fast, broad, frequent scans of traditional SAST. The rest of this guide explains what FAA does, how it works, how to install and run it, how to consume its results, and what to expect from it.

1.1 Changes since 26.4-preview5

This guide describes FAA version 26.4.0. The main changes relative to 26.4-preview5 are listed below. For the detailed history of 26.4-preview1 through 26.4-preview5, see the 26.4-preview5 edition of this guide.

- *Markdown output*: besides SARIF, a scan can write Markdown reports (a summary, a full report, or one document per finding), selected with `--output-format` (Section 4.5.5).
- *Converting existing results*: the new `-convert` command produces any output format from an existing SARIF, without LLM calls or cost (Section 4.5.6).
- *Output naming*: `--output` names all selected artifacts, and `--output-<format>` places an individual one. A run that would keep neither SARIF nor zip is refused unless `--no-sarif` is passed (Section 4.5).
- *Categories*: validation aligns categories more consistently: Fortify cate-

gories first, then a fixed set of agentic categories, with an explicit fallback for anything else. When two findings are merged as duplicates, the more severe one is kept.

- *Provenance*: the SARIF records the git commit, branch, and tags of the scanned tree (Section [8.1.3](#)).
- *Reliability*: a unit of work that shows no agent activity for 15 minutes is aborted, so the scan stops and can be resumed instead of hanging.
- *Policy file*: the experimental `analysis.prefetch.*` properties are removed (a policy that still sets them keeps working), and the shipped comments are shorter. Files with the extensions `min` and `fpr` were not excluded from analysis by the default policy; they now are.
- *Fixes*: masking a hardcoded secret no longer alters surrounding text that happens to contain the same word.
- *Documentation*: new sections explain how FAA structures its SARIF (Section [8.1](#)) and how to report a problem (Chapter [11](#)).

1.2 What Fortify Agentic Analyzer does

You run FAA against a local checkout of your application's source code. It drives an autonomous *coding agent* over the project: the agent explores the source tree the way a human security reviewer would (opening files, searching, following the call paths that matter), while the large language model (LLM) behind it forms and checks hypotheses about how the application could be attacked. The FAA harness turns this into a repeatable, unattended scan: it enumerates the project so that every relevant file is covered, has each candidate finding re-reviewed in a validation stage to weed out false positives, and emits the outcome as standard SARIF 2.1.0, ready to be consumed by Fortify on Demand, Fortify SSC, or any other SARIF-aware tool. For more information, see Chapter [8](#).

Because the analysis is driven by genuine code understanding rather than by deterministic rules, FAA finds issues that depend on grasping what the application is supposed to do: flawed authentication or authorization designs, business-logic weaknesses, subtle injection and persistence patterns, and other defects that do not present as a simple source-to-sink data flow. This is the class of issues that otherwise surfaces only in an expert manual code review. And because the agent needs no language-specific analyzer, this works on effectively any pro-

programming language, including languages for which Fortify SAST has limited or no support.

This deep, LLM-driven analysis has two defining characteristics:

- It can uncover vulnerabilities that traditional SAST misses, on any language.
- It has a real and significant operating cost, because every scan makes actual, paid LLM calls. A single full scan of a real business application can cost on the order of tens to hundreds of US dollars.

These characteristics shape how FAA is meant to be used: as a high-value, risk-based complement to Fortify SAST rather than as a check that runs on every commit. Combining the two analysis styles in this way is what we call hybrid SAST. For more information, see [Section 2.1](#).

1.3 Fortify Agentic Analyzer and Fortify AI-powered SAST

The Fortify portfolio already includes an [AI-powered SAST](#) capability, integrated into sourceanalyzer and generally available since the 26.1 release. Because both that feature and FAA use AI, it is worth being precise about how they differ: they are distinct capabilities that solve different problems. The following table summarizes the key differences.

Aspect	AI-powered SAST	Fortify Agentic Analyzer (FAA)
Depth	A basic level of AI-assisted analysis; less deep than the dedicated analysis Fortify performs on well-supported languages such as Java	Very deep scanning on any language; can uncover issues missed by traditional SAST, even on well-supported languages
Packaging	Integrated into source analyzer	A separate executable
Core technology	Non-agentic	Agentic
Bring-your-own LLM (non-FoD)	Yes, with a limited set of supported LLMs	Yes, with a very wide set of supported LLMs and providers
Bring-your-own LLM (FoD)	No, OpenText provides the LLM	Yes, same as in the non-FoD case
Operating cost	Relatively cheap to operate	Expensive to operate; significant token costs per scan (a full scan of a large application can cost hundreds of US dollars; Section 10.1 gives an example)

In short: AI-powered SAST adds a basic AI layer to broaden traditional scanning across many languages cheaply; FAA is a separate, agentic deep-analysis tool you run deliberately when you want to find what other techniques cannot. Both support a bring-your-own-LLM approach, but FAA supports a broader range of models and providers.

1.4 Where to start

- Understand how FAA works before you use it: [Chapter 2](#)
- Install FAA and set up its credentials: [Chapter 3](#)
- Run your first scan: [Chapter 4](#)
- Look up a command-line option or a policy property: [Chapters 5 and 6](#)
- Review the results in FoD or SSC: [Chapter 8](#)
- Estimate what a scan costs and how long it takes: [Chapter 10](#)
- Report a problem: [Chapter 11](#)

1.5 Conventions in this guide

Literal text (commands, options, file names, and property names and values) is shown in monospace, for example `fortifyaa -scan`. Angle brackets mark a value you supply: in `--output <path>`, replace `<path>` with your own path. Square brackets in a command synopsis mark an optional part, as in `-license [clear]`.

2 Concepts and architecture

This chapter explains the ideas behind FAA and how it is built. Read it before you run your first scan: understanding the hybrid SAST approach, the role of the coding agent and the LLM, and the three-stage analysis process makes the command-line options and configuration properties in later chapters much easier to reason about.

2.1 Hybrid SAST as a complementary approach

There are two broad styles of automated code analysis, and they have complementary strengths. Figure 2.1 summarizes how they compare.

Traditional SAST runs one or more deterministic algorithms over the code. Low-end tools do little more than simple pattern recognition; higher-end tools rely mainly on data-flow analysis, tracing tainted data from sources to sinks. Fortify SAST (formerly Fortify Static Code Analyzer) is exceptionally strong here in that it runs a suite of purpose-built analyzers in concert, each specialized for a different family of weaknesses. Data-flow analysis performs the source-to-sink taint tracing described above. Control-flow analysis reasons about the order in which operations execute. Semantic analysis recognizes dangerous uses of specific APIs. Structural analysis matches insecure code patterns and constructs. Configuration analysis inspects deployment descriptors and settings. Further analyzers address buffer, content, and encoding concerns. It is this breadth, not any single technique, that lets one scan cover a wide range of weakness classes at once. And because every analyzer is deterministic and rule- or model-driven (rather than reasoning turn by turn with an LLM), the combined scan is cheap to operate, highly deterministic, and fast enough to run daily or on every commit across an entire application. This is the “fast, broad, and frequent” foundation of an application security program.

Agentic analysis makes the opposite trade-off. It is built on an LLM that reads and reasons about the code much as an expert reviewer would. Its detection

completeness can exceed what a fixed set of analyzers achieves, especially for design-level and business-logic flaws. But it is comparatively expensive and only partially deterministic, so it is best run risk-based and infrequently (weekly or less) on the applications and changes that warrant it.

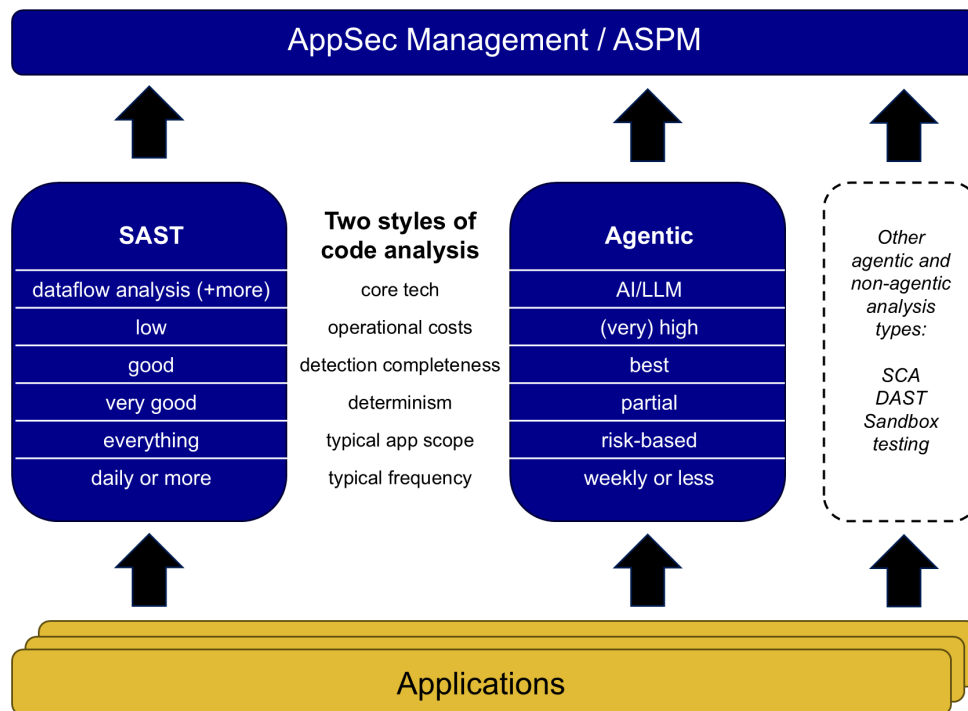


Figure 2.1: Two complementary styles of code analysis, feeding a single AppSec management platform.

Agentic code analysis has clear added value above traditional SAST. But because of its cost, it is not a credible replacement for SAST: running it at the frequency and breadth that SAST runs at would be prohibitively expensive. The right model is therefore a hybrid one, in which agentic analysis is deployed as a complementary technology alongside SAST, software composition analysis (SCA), dynamic application security testing (DAST), and other analysis types, all feeding a single AppSec management platform (FoD or Fortify SSC) that manages applications, deduplicates, and reports on compliance centrally.

In practice this means:

- *Normally, you run a traditional Fortify SAST scan first, then run FAA and give it the SAST results. FAA uses those results to filter out anything SAST already found, so it reports only genuinely complementary findings (Section 4.4).*

- *You can also run FAA on its own*, for example on a language that traditional SAST does not support, as an FAA-only scan. This is fully supported; it simply forgoes the deduplication against existing SAST results.

The high per-scan cost is the central reason a “go fully agentic” strategy is unattractive, and the reason FAA is positioned as a deliberate, risk-based complement rather than an everyday gate.

2.2 General architecture

FAA is best understood as four layers, each adding something the layer below it cannot provide on its own. Figure 2.2 shows the layers, and the list after it describes each one.

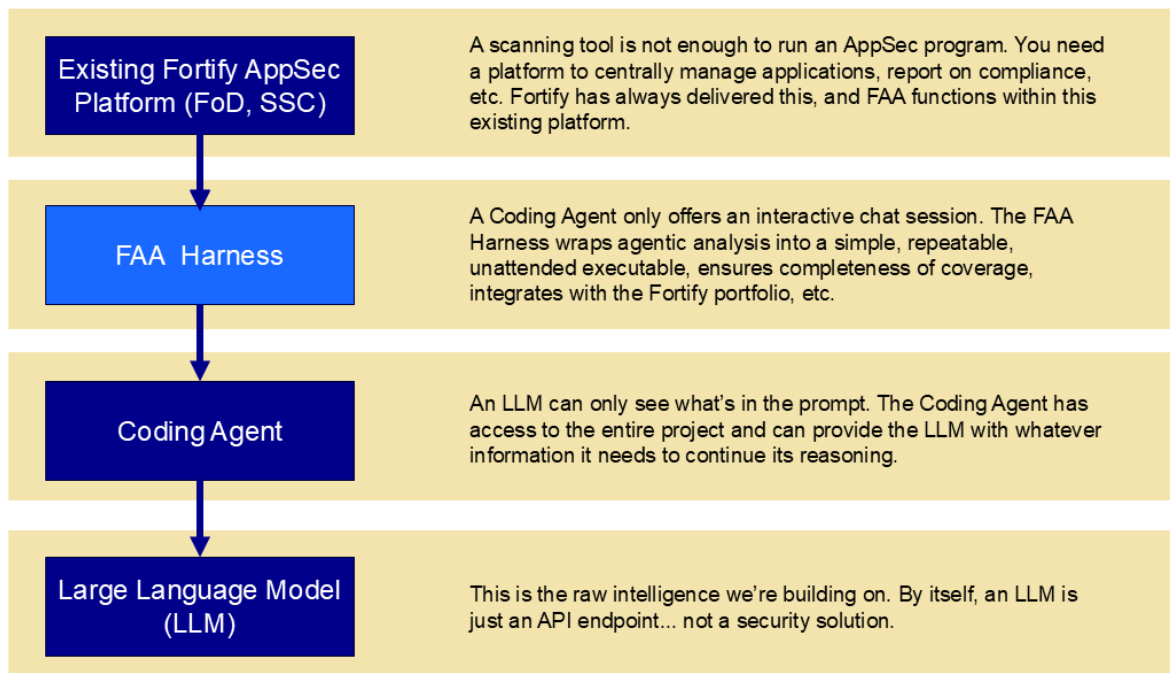


Figure 2.2: The four layers of FAA, from the raw LLM up to the Fortify platform.

Large Language Model (LLM). The raw intelligence FAA builds on. By itself, an LLM is just an API endpoint: it can only see what is placed in its prompt, and it is not a security solution.

Coding Agent. Wraps the LLM with the ability to explore a project. The agent has access to the entire source tree and can feed the LLM whatever infor-

mation it needs to continue its reasoning (reading files, searching, listing directories). On its own, however, a coding agent only offers an interactive chat session.

FAA Harness. The executable you run. It turns interactive agentic analysis into a simple, repeatable, unattended tool: it enumerates the project, guarantees complete coverage, drives the agent through the analysis process, applies all deterministic post-processing, and integrates with the Fortify portfolio. Everything reproducible about a scan lives here, not in the agent.

Existing Fortify AppSec Platform (FoD / SSC). A scanning tool is not enough to run an AppSec program; you need a platform to manage applications, report on compliance, and track findings over time. Fortify has always provided this, and FAA's results flow into it like any other scan type.

2.3 Communication architecture

During a scan, the FAA harness coordinates the coding agent and, through it, the LLM. Figure 2.3 shows how these components communicate with each other and with the filesystem.

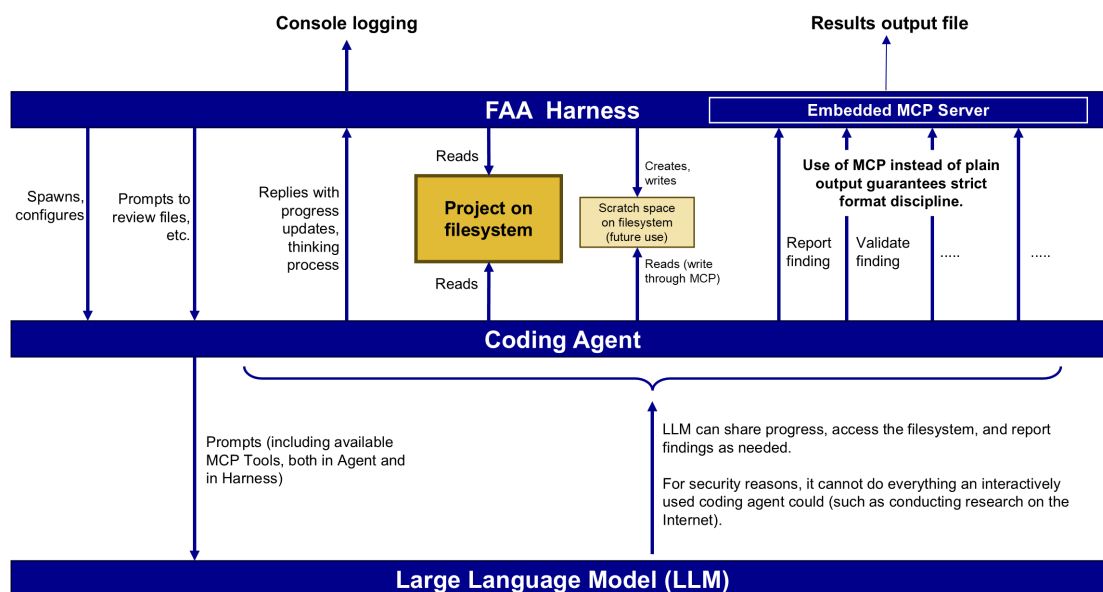


Figure 2.3: Communication channels between the harness, the agent, the LLM, and the filesystem.

The harness spawns and configures the coding agent and sends it prompts (for example, “review these files”). The agent prompts the LLM, streams back progress updates and its thinking process, and reads the project from the filesystem. FAA treats the project as read-only: it never modifies the code under analysis.

The figure also shows a separate scratch space on the filesystem. This is the agent’s isolated working directory, and it doubles as an isolation boundary. Because the agent’s working directory is an empty temporary folder rather than the project itself, the agent does not discover instruction files in the scanned tree (for example a `CLAUDE.md` or `AGENTS.md`), and they cannot redirect it. The figure labels the space “future use”: in the current release the agent has no write tool (Chapter 9), so it does not write working files there, and the harness does not yet exchange files with the agent through it. The intent is that a future, dedicated structured tool could let the agent keep working notes in this space, writing only through that tool, without enabling general file writes.

Crucially, the agent does not emit results as free-form text. The harness exposes an embedded Model Context Protocol (MCP) server offering a small set of schema-validated structured tools, such as “report finding” and “validate finding.” The agent reports findings by calling these tools. Using MCP rather than plain output guarantees strict format discipline: every finding arrives as well-formed, validated data, which is what makes FAA’s output reproducible and machine-consumable. The harness also writes ordinary console logging and, at the end of the run, the results output file.

For security reasons, FAA deliberately does not give the agent everything an interactively used coding agent could do. In particular, the agent cannot write to the project or reach the Internet. It can read the code, share progress, and report findings, and nothing more. Chapter 9 describes these guarantees and their limits in detail.

2.4 Coding agent options

FAA can drive either of two coding agents behind a single abstraction layer in its source code, as Figure 2.4 shows. You choose one with the `agent` property in the policy file (Chapter 6).

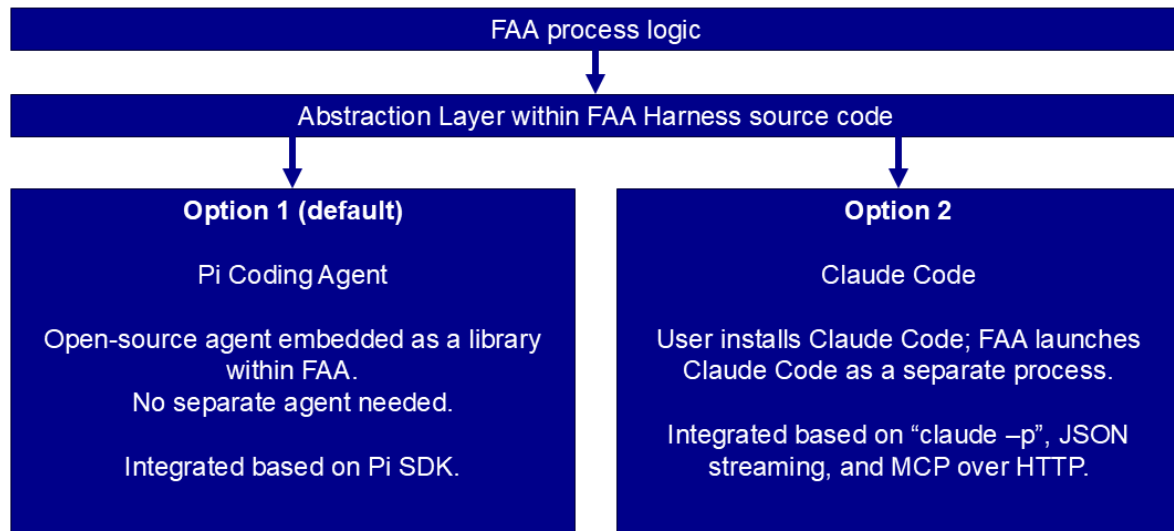


Figure 2.4: Two interchangeable coding-agent backends behind one abstraction layer.

Pi (the default). An open-source coding agent embedded directly inside FAA as a library, integrated via the Pi SDK. Nothing separate needs to be installed; the bundled agent runs in-process. This is the recommended option for most users.

Claude Code. You install Anthropic’s Claude Code CLI separately, and FAA launches it as a child process (integrated via `claude -p`, JSON streaming, and MCP over HTTP). Choose this if you already use and trust Claude Code and prefer to drive your analysis through your existing installation and authentication.

The following table compares the two options.

Aspect	Pi (default)	Claude Code
What to install	Nothing; Pi is bundled in the FAA executable	Anthropic’s <code>claude</code> CLI, logged in and on the PATH
LLM access	Credentials you provide for a model provider (Section 3.7)	Claude Code’s own authentication

Continued on next page

Aspect	Pi (default)	Claude Code
Models	A wide range of models and providers	Anthropic's Claude models
Billing	By your model provider	However your Claude Code use is billed
Subscriptions	GitHub Copilot only (Section 3.7.3)	Claude Pro and Max
Recommended for	Most users	Users who already work with Claude Code

Which models are available depends on the selected agent. Models are configurable per analysis stage (Section 2.5 and Chapter 6); strong models such as Anthropic's Sonnet and Opus are typical choices. For more information about configuring models and LLM access, see Chapter 3.

2.5 The analysis process

A scan runs as a pipeline of up to three LLM stages, all orchestrated by the harness. Figure 2.5 shows the pipeline, and the list after it describes each stage.

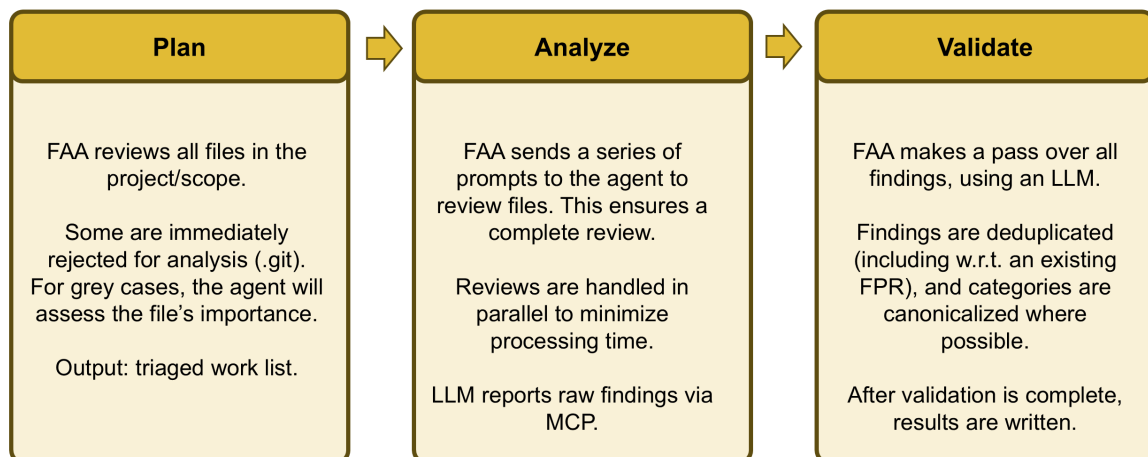


Figure 2.5: The three-stage analysis pipeline: Plan, Analyze, Validate.

Plan. FAA first walks every file in the project (or the chosen scope). A deterministic filter immediately drops files that are never worth analyzing: version-

control metadata such as .git, dependency and build directories, binaries, and so on. For the remaining “grey” cases, an optional LLM pass assesses each file’s importance and prioritizes the work. The output is a triaged work list.

Analyze. FAA sends the agent a series of prompts to review the in-scope files. The harness, not the agent’s memory, guarantees that every file is covered. Reviews run in parallel to minimize wall-clock time, and the agent reports raw findings via the MCP “report finding” tool.

Validate. FAA makes a pass over all findings using an LLM. Findings are deduplicated (including against any existing Fortify results you supplied), false positives are dropped, and categories are aligned to the canonical Fortify taxonomy where possible. Only after validation completes are the results written.

An external policy file controls the parameters of this process: which LLM is used for each stage, the thinking effort, the maximum cost, batch sizes, concurrency, and which stages run at all (Chapter 6). Each stage can be enabled or disabled individually: for example, you can run a cheap plan-only diagnostic, or skip validation to save cost at the expense of output quality.

2.6 SARIF output and the Fortify taxonomy

FAA’s sole output contract is SARIF 2.1.0, a standard, tool-neutral format that Fortify and many other tools can read. The harness, not the agent, owns all of the formatting, so the output is deterministic and reproducible regardless of how the agent phrased things internally. Key points of the mapping:

- *Each finding is a SARIF result with its own rule.* The finding’s detail (its explanation and its remediation guidance) is written for that specific piece of code, and SARIF only has somewhere to put such text on the rule, so FAA emits one rule per finding. Section 8.1 explains this in full. Each finding also carries a single canonical sink location with a short code snippet.
- *Categories are aligned to the Fortify taxonomy* during validation. The taxonomy is intentionally not exhaustive, so a genuine issue that falls outside it

is preserved under an `agentic/` rule namespace while keeping its human-readable category name.

- *Severity* is emitted both as a standard SARIF level and as a Fortify-specific severity property that SSC treats as authoritative.
- *Stable fingerprints* give each finding a durable instance identifier, so the same issue is recognized across re-scans.

Because the output is standard SARIF, the same file is consumable by Fortify on Demand (natively) and Fortify SSC (via a parser plugin). Chapter 8 covers each consumer. For complementary scans, the findings that existing Fortify SAST already reported are dropped before the SARIF is written, so what you upload is only what SAST did not already find.

3 Installation and setup

FAA ships as a single native executable plus a set of policy files. Setting it up means deciding where you will run it, putting those files in place, choosing which coding agent drives the analysis, and providing two kinds of credentials: a Fortify entitlement (so FAA can verify your license) and LLM access (so it can do the actual analysis). This chapter covers each step, starting with what to know before you install and ending with how to verify the whole setup.

3.1 The local execution model

FAA always runs locally, on your own workstation or your own CI/CD runner, directly against a checkout of the source code. This is true even if you use Fortify on Demand (FoD).

This deserves emphasis, because it is at odds with how FoD normally works. FoD is a SaaS service: you typically upload code or artifacts and the analysis runs in OpenText's cloud. FAA does not work that way. Two properties of agentic analysis make local execution necessary:

- *The agent needs direct, read access to the entire source tree* to explore and reason about it. The analysis is intrinsically code-resident.
- *LLM usage is bring-your-own and billed to you* by your LLM provider. Running locally is what lets you use your own LLM access and control your own spend (Section 3.7 and Chapter 10).

So regardless of whether you ultimately consume results in FoD or in SSC, the FAA scan itself happens locally: you run the executable where the code is, it produces a SARIF file, and you then bring that result into your Fortify platform (Chapter 8).

3.2 Prerequisite: Fortify CLI (fcli)

Two FAA options read data live from your Fortify platform, and they do so by invoking the Fortify CLI (fcli). FAA expects to find fcli on your PATH:

- `--fod-release`: reads the existing issues of an FoD release, so the scan reports only what SAST did not already find (Section 4.4)
- `--ssc-appversion`: the same, for an SSC application version

FAA does not install fcli for you. That is a deliberate choice rather than an omission: fcli is a separate Fortify component with its own release cadence and its own installation instructions, and FAA relates to it the same way the rest of the Fortify portfolio does. Install it once, yourself, from <https://github.com/fortify/fcli>.

Before you run a scan that uses either option, check that fcli is ready:

```
fcli --version          # fcli is installed and on the PATH
fcli fod session list   # an FoD session is logged in
fcli ssc session list   # or, for SSC, an SSC session
```

If the `fcli` command is not found, install fcli or add its directory to the PATH. If no session is listed, or the session has expired, log in with `fcli fod session login` or `fcli ssc session login`. For example, to log in to FoD with API credentials and then run a complementary scan against a release:

```
fcli fod session login --url https://ams.fortify.com \
  --client-id <id> --client-secret <secret>
fortifyaa -scan . --fod-release MyApp:1.0
```

If one of these options is used and fcli cannot be found, FAA reports a clear error naming the missing dependency rather than failing obscurely later.

Everything else in FAA works without fcli. In particular, the license gate (Section 3.5) does not call fcli: it reads your credentials straight from the environment. It uses fcli's variable names purely so that a machine or pipeline already configured for fcli needs no additional setup.

3.3 Installers

FAA is delivered as an *installer* for each platform, which places the native executable and the shipped policy file in an installation directory of your choosing (Section 3.4). The following table lists the installers.

Platform	Installer
Windows (x64)	OpenText_FAA_windows-x64.exe
Linux (x64)	OpenText_FAA_linux-x64.run
Linux (ARM64)	OpenText_FAA_linux-arm64.run
macOS (Apple silicon)	OpenText_FAA_osx.app.zip, a zip archive holding the OpenText_FAA_osx.app installer application

The executable itself is named `fortifyaa` (`fortifyaa.exe` on Windows) and is built natively per platform. There is no macOS (Intel) build, and none is planned. Installers and executables are code-signed: the Windows build with OpenText's Authenticode certificate, the macOS build with OpenText's Apple Developer ID and notarized, so both run as installed without security prompts.

3.4 Install Fortify Agentic Analyzer

The installer is deliberately minimal. It does four things:

- copies the executable and `fortify-aa-default.properties` into the installation directory, and writes nothing anywhere else;
- adds the installation directory to the `PATH` (Section 3.4.3);
- places an uninstaller in the installation directory (Section 3.4.4) and, on Windows, lists FAA under *Installed apps* so it can be removed from there;
- writes an installation log, `OpenTextFAA-<version>-install.log`, to the system temporary directory: `%TEMP%` on Windows, `/tmp` on Linux and macOS (always `/tmp`; `TMPDIR` is not consulted). When an installation fails, this is the place to look. The log is readable by the installing user only, so when a different user installed earlier on the same machine the named log cannot be replaced and the new run's log appears as `installbuilder_installer_<`

n>.log in the same directory instead.

On Linux, both the installer and FAA itself are built against glibc; musl-based distributions such as Alpine are not supported.

Every release installs into its own directory, named after the release label, so two versions never overwrite each other. The default location depends on the platform and on whether the installer runs with administrative (root) privileges:

Platform	Administrative install	Non-administrative install
Windows	C:\Program Files\Fortify\OpenText_FAA_<version>	C:\Fortify\OpenText_FAA_<version> (interactive installs only; see Section 3.4.2)
Linux	/opt/Fortify/OpenText_FAA_<version>	~/Fortify/OpenText_FAA_<version>
macOS	/Applications/Fortify/OpenText_FAA_<version>	~/Fortify/OpenText_FAA_<version> (interactive installs only; see Section 3.4.2)

Whatever the location, the result is a self-contained directory:

```
fortifyaa(.exe)
fortify-aa-default.properties
Uninstall_OpenTextFAA(.exe / .app)
Uninstall_OpenTextFAA.dat
... any additional fortify-aa-<name>.properties policies you add ...
```

Running the installer again with the same directory is harmless: it replaces the shipped files in place and leaves anything you added there untouched.

Keep the policy files next to the executable. Configuration is never built into the binary. At startup FAA reads its policy file (by default `fortify-aa-default.properties`) from its own directory, so the shipped `fortify-aa-*.properties` files must sit alongside `fortifyaa`; custom policies (Chapter 6) go in the same place. The installer is a convenience, not a requirement: because the directory is self-contained, you can equally copy its contents by hand to wherever FAA should run (into a CI runner image, for example) and run it from there.

3.4.1 Interactive installation

The interactive installer is a short wizard: it shows the license agreement, asks for the installation directory, and installs.

Windows — run `OpenText_FAA_windows-x64.exe`. The installer asks for the highest privilege level your account can obtain, so an administrator account gets a User Account Control (UAC) prompt and installs under Program Files; a standard account runs without elevation and is offered `C:\Fortify` instead.

Linux — make the file executable, then run it. With a graphical display available the wizard opens in a window; without one (no `DISPLAY`, as on a server or over plain SSH) it falls back, silently, to a text-mode wizard in the terminal, which you can also request explicitly with `--mode text`. The text wizard pages through the license agreement (*Enter* for the next page), asks *Do you accept this license? [y/n]*, asks for the installation directory with the default shown in brackets, asks for a final confirmation, and installs. Run it with `sudo` to install under `/opt`, or as yourself to install under your home directory; FAA needs no root privileges to run.

```
chmod +x OpenText_FAA_linux-x64.run
./OpenText_FAA_linux-x64.run
```

The text wizard needs a real terminal to read from. If its standard input is closed or redirected (as happens in a pipeline that forgets `--mode unattended`), it does not fail but re-issues its prompt forever, flooding the output. In any script, always pass `--mode unattended` (next section).

macOS — unzip `OpenText_FAA_osx.app.zip` and open the `OpenText_FAA_osx.app` it contains, from the Finder or with `open`. The installer application starts without administrator rights, so the wizard proposes a directory in your home, `~/Fortify/OpenText_FAA_<version>`. FAA needs no special privileges to run. For a machine-wide install, enter `/Applications/Fortify/OpenText_FAA_<version>` instead, from an administrator account.

3.4.2 Unattended installation

For servers, CI/CD runners, and image builds, run the installer in *unattended mode*: `--mode unattended`. It then shows no wizard and asks no questions: the license agreement that the interactive wizard displays is accepted implicitly, every setting takes its default, and the installer exits with status 0 on success or status 1 on failure. On success it prints nothing at all. On failure it prints a short reason, for example “Unable to create directory”, on its standard output rather than standard error, so capture both; the installation log has the details. It runs fine without a terminal, so it can be called from any pipeline step or Dockerfile RUN line.

```
# Windows -- from an elevated PowerShell or Command Prompt (see below)
OpenText_FAA_windows-x64.exe --mode unattended
OpenText_FAA_windows-x64.exe --mode unattended --installdir "D:\Fortify\FAA"

# Linux -- with sudo for /opt, or as yourself with an explicit directory
chmod +x OpenText_FAA_linux-x64.run
sudo ./OpenText_FAA_linux-x64.run --mode unattended
./OpenText_FAA_linux-x64.run --mode unattended --installdir "$HOME/faa"

# macOS -- run the script inside the installer application; the unattended
# default is /Applications, so without administrator rights give a directory
# in your home
unzip OpenText_FAA_osx.app.zip
sudo OpenText_FAA_osx.app/Contents/MacOS/installbuilder.sh --mode unattended
OpenText_FAA_osx.app/Contents/MacOS/installbuilder.sh --mode unattended \
  --installdir "$HOME/Fortify/OpenText_FAA_<version>"
```

The options that matter for unattended use (`--help` lists them all):

Option	Meaning
<code>--mode unattended</code>	Install without a wizard. The other values are the interactive modes: win32 (Windows), osx (macOS), gtk and xwindow (Linux, graphical; gtk is the Linux default), and text (Linux and macOS, in the terminal).
<code>--installdir <dir></code>	The installation directory. Defaults as in the table above; the directory is created if it does not exist (parents included) and must be writable by the installing user. A relative path is resolved against the current directory; spaces are fine when quoted.
<code>--unattendedmodeui <ui></code>	What, if anything, is displayed while installing unattended: none (the default: nothing at all), minimal (a progress window), or minimalWithDialogs (progress plus any dialogs, which may then require interaction). Leave it at none on a server.
<code>--optionfile <file></code>	Read installation settings from a file instead of the command line, one key=value per line, for example a single line <code>installdir=/opt/Fortify/OpenText_FAA_<version></code> . Useful when the same settings are applied on many machines.
<code>--debuglevel <0..4>, --debugtrace <file></code>	Raise the verbosity of the installation log (default level 2) and, with <code>--debugtrace</code> , write a detailed trace to <file>. Use these when diagnosing a failing installation.
<code>--version, --help</code>	Print the product version, or the option list, and exit.

Windows: elevation and console output. Two properties of the Windows installer matter when it is run from a script.

- *It requests elevation.* The installer is marked to run with the highest privileges the account can obtain. For an administrator account that means a UAC prompt, also in unattended mode and even for `--help`, and launching it from a non-elevated prompt fails outright with “The requested operation requires elevation”. Run it from a context that is already elevated, such as a PowerShell started with *Run as administrator* or a CI agent or service that runs elevated; no prompt appears then. A standard (non-administrator) account runs the installer without elevation; because the default directory under Program Files is then not writable, and the interactive wizard’s fallback to `C:\Fortify` does not apply in unattended mode, pass `--installdir` explicitly.
- *It is a GUI application.* A Command Prompt or PowerShell console shows no output from it and does not wait for it to finish. To run it synchronously and check the result, use `start /wait` in a batch file, or `Start-Process -Wait` in PowerShell:

```
$p = Start-Process -FilePath .\OpenText_FAA_windows-x64.exe `
    -ArgumentList '--mode','unattended' -Wait -PassThru
if ($p.ExitCode -ne 0) { throw "FAA install failed: $($p.ExitCode)" }
```

Verifying. The PATH change (next section) is not visible in the shell that ran the installer, so verify with the full path, at no cost:

```
/opt/Fortify/OpenText_FAA_<version>/fortifyaa -selftest --no-live
```

3.4.3 The PATH entry

The installer adds the installation directory to the PATH so that `fortifyaa` can be typed from anywhere: on Windows through the registry (the system-wide PATH for an elevated install), on Linux and macOS by appending a line of the form

```
export PATH="/opt/Fortify/OpenText_FAA_<version>:$PATH"
```

to the shell startup files in the home directory of the user running the installer (the directory named by `HOME`): on Linux, `~/.bashrc` and `~/.profile`; on macOS, `~/.bashrc` and `~/.bash_profile` for `bash` and `~/.zprofile` for `zsh`, the default shell,

created if it does not exist yet. Nothing else is touched; other shells need the line added by hand. Three consequences are worth knowing before relying on it:

- It takes effect in new shells (on Linux and macOS, new login shells), not in the one that ran the installer.
- On Linux and macOS it is per user. An install done with `sudo` edits root's shell files, not yours, and a service account that runs scans never sees it.
- Pipelines should not depend on it. Invoke FAA by its full path, or add the installation directory to the `PATH` yourself in the pipeline's own configuration.

3.4.4 Upgrade and uninstall

Because every release has its own directory, an upgrade is simply installing the new release and then uninstalling the old one; the two never overwrite each other. Scan state and stored credentials live in the Fortify working directory (Sections 4.6 and 3.7.2), not in the installation directory, so neither is affected by upgrading or uninstalling.

Before uninstalling, move anything you added to the installation directory. The uninstaller removes everything in it, including custom policy files, `pi_models.json`, and `pi_extensions/` (Chapter 7). It then removes the `PATH` entry, and the enclosing Fortify directory if that is now empty. It writes `OpenTextFAA-<version>-uninstall.log` next to the installation log.

The uninstaller sits in the installation directory as `Uninstall_OpenTextFAA.exe` (Windows; also reachable from *Installed apps*), `Uninstall_OpenTextFAA` (Linux), or `Uninstall_OpenTextFAA.app` (macOS). It always removes the installation it was created in (there is no `--installdir` option) and takes the same `--mode` option as the installer. Interactively it asks a single *Do you want to uninstall ...? [Y/n]* question; with `--mode unattended` it asks nothing, prints nothing, and exits with status 0 on success:

```
# Windows (elevated, if the install was)
Uninstall_OpenTextFAA.exe --mode unattended

# Linux
```

```
sudo ./Uninstall_OpenTextFAA --mode unattended

# macOS
sudo Uninstall_OpenTextFAA.app/Contents/MacOS/installbuilder.sh --mode
  ↪ unattended
```

3.5 Fortify entitlement (the license gate)

FAA verifies that you have a valid Fortify entitlement before it does any analysis or incurs any LLM cost. You satisfy this gate by providing either FoD or SSC credentials in the environment. The two are mutually exclusive: provide one set.

The variable names are fcli's, so a machine or pipeline already configured for fcli needs no additional setup: fcli lets any option be defaulted from the environment as `FCLI_DEFAULT_<PLATFORM>_<OPTION>`, and FAA reads exactly those.

Fortify on Demand — set `FCLI_DEFAULT_FOD_CLIENT_ID` and `FCLI_DEFAULT_FOD_CLIENT_SECRET`, or alternatively `FCLI_DEFAULT_FOD_USER`, `FCLI_DEFAULT_FOD_PASSWORD`, and `FCLI_DEFAULT_FOD_TENANT`. By default FAA contacts the AMS (Americas) FoD instance; if your tenant is hosted elsewhere, also set `FCLI_DEFAULT_FOD_URL` to your regional FoD URL. Either the portal form (for example, <https://emea.fortify.com>) or the API form (for example, <https://api.emea.fortify.com>) works.

Fortify SSC — set `FCLI_DEFAULT_SSC_URL` and `FCLI_DEFAULT_SSC_TOKEN`, or alternatively `FCLI_DEFAULT_SSC_URL` with `FCLI_DEFAULT_SSC_USER` and `FCLI_DEFAULT_SSC_PASSWORD`.

Older variable names. Releases before 26.4-preview4 used a shorter, FAA-specific set of names (`FOD_CLIENT_ID`, `SSC_URL`, and so on, without the `FCLI_DEFAULT_` prefix). Those are still honored, so existing pipelines keep working unchanged; if both spellings are set, the `FCLI_DEFAULT_` one is used. New setups should use the `FCLI_DEFAULT_` names. Migrating an existing one is a rename:

Older name	Use instead
FOD_URL, FOD_TENANT, FOD_USER, FOD_PASSWORD, FOD_CLIENT_ID, FOD_CLIENT_SECRET	The same name prefixed with FCLI_DEFAULT_
SSC_URL, SSC_TOKEN, SSC_USER, SSC_PASSWORD	The same name prefixed with FCLI_DEFAULT_

The result of the check is cached for 24 hours so that repeated runs do not re-verify every time. You can verify access at any time, and reset the cache, with the `-license` command (Chapter 5):

```
fortifyaa -license          # verify Fortify access (uses the 24h cache)
fortifyaa -license clear    # reset the cached result
```

3.6 Set up the coding agent

FAA does the analysis by driving one of two coding agents over your source tree; Section 2.4 introduced the two and when to choose which. The choice is made with the `agent` property in the policy file (Chapter 6), and it determines both what to install and how you provide LLM access:

Pi (the default, `agent=pi`) — bundled inside the FAA executable and run in-process; there is nothing to install. You bring your own LLM access (credentials for a model provider such as AWS Bedrock, Google Vertex AI, Azure OpenAI, or GitHub Copilot), as described in Section 3.7, and that provider bills you for LLM usage.

Claude Code (optional, `agent=claude`) — FAA launches Anthropic’s separately installed `claude` CLI as a child process and uses its existing authentication. Install the `claude` CLI yourself, log it in, ensure it is on your `PATH`, and set `agent=claude` in your policy. You do not configure any LLM credentials in FAA itself: the LLM access configuration in Section 3.7 applies only to the default, bundled Pi agent. This is also the way to run FAA on a Claude Pro or Max subscription. Claude Code is Anthropic’s first-party tool, so the sub-

scription applies there normally, unlike with the Pi agent, where Anthropic does not permit it (Section 3.7.3).

3.7 LLM access for the Pi agent (bring your own LLM)

Everything in this section applies to the default, bundled Pi agent. If you use `agent=claude`, skip ahead to Section 3.8. FAA supports a wide range of models and providers, and there are two ways to give it access: reuse each provider's own standard credential setup from the environment (Section 3.7.1, the default), or store a credential (a GitHub Copilot subscription sign-in or a provider API key) once with the interactive `fortifyaa -pillogin` command (Section 3.7.2). Section 3.7.3 explains why Copilot is the only subscription offered, and Section 3.7.4 how FAA decides which provider and model a scan then actually uses.

3.7.1 Credentials from the environment

The Pi agent authenticates to each provider exactly the way that provider's own CLI and SDKs do: it reads a few environment variables to select an account, project, or region, and picks up the actual credentials from the standard place for that cloud (for example `~/.aws` for AWS, or the `gcloud` Application-Default-Credentials file for Google). The practical rule of thumb: if the provider's own tooling already works in your shell, FAA will use it too.

AWS Bedrock — FAA uses the standard AWS SDK credential chain, so you configure it the same way as the AWS CLI. The most common setup is to select a named profile with `AWS_PROFILE` (use `default` for the default profile) and set `AWS_REGION`; the credentials themselves stay in `~/.aws/credentials` and `~/.aws/config` (including SSO, `credential_process`, or assumed-role setups). Alternatively, put credentials directly in the environment (`AWS_ACCESS_KEY_ID` + `AWS_SECRET_ACCESS_KEY`, optionally `AWS_SESSION_TOKEN`), supply a Bedrock bearer token (`AWS_BEARER_TOKEN_BEDROCK`), or rely on an EC2/ECS/EKS instance role. At least one of these must be present for FAA to detect Bedrock. In particular, a `~/.aws` profile is only picked up when

you also set `AWS_PROFILE`. When several cross-region inference profiles of a model are available, FAA prefers the one matching `AWS_REGION`.

Google Vertex AI — FAA uses Google Application Default Credentials (ADC), the same mechanism as the `gcloud` CLI. Authenticate once with `gcloud auth application-default login` (which writes `~/.config/gcloud/application_default_credentials.json`), or point `GOOGLE_APPLICATION_CREDENTIALS` at a service-account key file; then set `GOOGLE_CLOUD_PROJECT` and `GOOGLE_CLOUD_LOCATION` (for example `us-central1`). All three (credentials, project, and location) must be present for FAA to detect Vertex.

Azure OpenAI — Azure is key-based: set `AZURE_OPENAI_API_KEY` together with your endpoint, given as either `AZURE_OPENAI_BASE_URL` or `AZURE_OPENAI_RESOURCE_NAME`. `AZURE_OPENAI_API_VERSION` and `AZURE_OPENAI_DEPLOYMENT_NAME_MAP` (which maps model ids to your Azure deployment names) are optional.

Direct API keys — providers such as Anthropic and OpenAI read a single API-key environment variable (`ANTHROPIC_API_KEY`, `OPENAI_API_KEY`). Pi supports many more providers besides these.

In every case FAA resolves these credentials from your existing environment and cloud-SDK setup; it never copies them into a FAA-specific file, and it never reads Pi's own `~/.pi` directory.

To see exactly which providers and models your current setup makes available, run `fortifyaa -selftest` (Section 3.8); it lists each usable model as `provider/id`. The full catalog of providers and models, and each provider's setup details, are documented at <https://pi.dev/models> and <https://pi.dev/docs/latest/providers>.

3.7.2 Store credentials with `-piloin`

Instead of configuring the environment, you can store a credential once with the interactive login command (the “pi” in the name reflects that it applies to the bundled Pi agent; the Claude Code agent brings its own `claude` login):

```
fortifyaa -piloin           # choose subscription or API key
fortifyaa -piloin status    # show what is stored (no secrets)
fortifyaa -piloin clear [<provider>] # remove one credential, or all
```

The command supports two styles:

Subscription sign-in (GitHub Copilot) — authenticate with your GitHub Copilot subscription via a device code. FAA stores the resulting tokens and refreshes them automatically; there is no API key to manage. Copilot is deliberately the only subscription FAA offers (Section 3.7.3).

API key — paste a provider API key once (for example an Anthropic or OpenAI key) instead of exporting it in every shell.

3.7.3 Subscription policies: why only GitHub Copilot

Whether a consumer AI subscription may be consumed by a tool other than the vendor’s own is each vendor’s decision, and in 2026 the three major vendors diverged sharply. FAA’s subscription menu reflects that landscape:

GitHub Copilot: permitted and metered. GitHub explicitly supports third-party agents and clients on paid Copilot plans¹. Usage is metered in “premium requests” included in each plan. Premium models such as Claude consume requests at a per-model multiplier, and once the monthly allowance is exhausted, overage is opt-in and billed per request². In short: the subscription genuinely carries the usage.

Anthropic (Claude Pro/Max): prohibited. Since February 2026, Anthropic’s terms state that “the use of OAuth tokens obtained via Claude Free, Pro, or Max accounts in any other product, tool, or service is not permitted”³, with enforcement completed in April 2026⁴. A Claude account signed into a third-party tool is billed from separately purchased usage credits; the

¹About third-party coding agents, GitHub Docs, <https://docs.github.com/en/copilot/concepts/agents/about-third-party-coding-agents>; GitHub also publishes a Copilot SDK exposing the same agent runtime, <https://github.com/github/copilot-sdk>.

²GitHub Copilot plans & pricing, <https://github.com/features/copilot/plans>.

³Anthropic clarifies ban on third-party tool access to Claude, The Register, 2026-02-20, https://www.theregister.com/2026/02/20/anthropic_clarifies_ban_third_party_claude_access/.

⁴Anthropic cracks down on unauthorized Claude usage by third-party harnesses, VentureBeat, <https://venturebeat.com/technology/anthropic-cracks-down-on-unauthorized-claude-usage-by-third-party-harnesses>.

subscription allowance itself never applies. FAA therefore does not offer Claude subscription sign-in. To use a Claude subscription with FAA, use the Claude Code agent instead: install Anthropic's `claude` CLI, log it in, and set `agent=claude` in your policy (Section 3.6). Claude Code is Anthropic's first-party tool, so the subscription applies normally there. For pay-per-use access with the Pi agent, use an Anthropic API key, or consume Claude through AWS Bedrock or Google Vertex AI.

OpenAI (ChatGPT Plus/Pro): not supported. ChatGPT subscriptions include Codex usage in OpenAI's own Codex products (the Codex CLI, IDE extension, and Codex cloud)⁵. Use of that sign-in by other tools is not officially supported⁶ and can stop working without notice, as both Anthropic and Google (Gemini CLI) demonstrated in 2026. FAA does not offer it. To use OpenAI models, use an OpenAI API key (billed at standard API rates) or Azure OpenAI.

Stored credentials live in FAA's own credential store at `~/.fortify/faa26.4/pi-auth.json` (`%LOCALAPPDATA%\Fortify\faa26.4\pi-auth.json` on Windows), overridable with `FORTIFYAA_PI_AUTH_PATH`. A stored credential takes precedence over environment variables for that provider; use `-piloin status` to see what is stored, and `-piloin clear` to return to the environment-based setup. AWS Bedrock and Google Vertex AI are not covered by `-piloin`; they use your ambient cloud-SDK credentials as described in Section 3.7.1. After logging in, verify with `fortify aa -selftest`: with `pi.provider=auto` (the default) the stored login is preferred automatically; pin `pi.provider` to be fully explicit (Section 3.7.4).

3.7.4 Choose the provider and model

Once one or more providers are configured (from the environment, from `-piloin`, or both), the policy file determines what a scan actually runs on: `pi.provider` controls which provider serves the requests, and the `*.model.*` properties control which model each analysis stage uses.

⁵Authentication — Codex, OpenAI, <https://developers.openai.com/codex/auth>.

⁶See, for example, the discussion in OpenAI's own Codex repository, <https://github.com/openai/codex/discussions/8338>; OpenAI has permitted specific third parties case by case, <https://thenextweb.com/news/openai-openclaw-chatgpt-subscription-agent>.

The provider (`pi.provider`)

The shipped default is `pi.provider=auto`. At the start of every scan (and of `-selftest`), FAA determines which providers offer the configured model family and probes them with a single tiny LLM call. Providers you signed into with `-pilogin` are tried first, then providers detected from the environment (alphabetical within each group, so the order is deterministic). The first provider that actually answers is used and stays fixed for the entire scan; FAA never switches providers mid-scan, and it reports its choice at startup. This makes `auto` robust against a configured-but-broken credential: an expired AWS session, for example, fails the probe immediately and FAA moves on to the next candidate instead of failing mid-analysis.

Note that only providers offering the configured model family are candidates. With the default policy the model families are Claude models, so, for example, an OpenAI login is not a candidate: logging in with `-pilogin` does not by itself make a provider eligible. FAA points this out at startup (“openai has stored `-pilogin` credentials but offers no ‘sonnet’ model”); to use such a provider, point the `*.model.family` policy properties at families it serves.

Alternatively, set `pi.provider` to a concrete provider id, such as `amazon-bedrock`, `google-vertex`, `azure-openai-responses`, `anthropic`, `github-copilot`, or `openai` (use the ids shown by `-selftest`), to make runs fully explicit and reproducible. A pinned provider is still probed at startup, so a dead credential fails immediately with a clear message, but FAA never substitutes a different provider for an explicit pin.

The model, per analysis stage

Which model each analysis stage uses is set in the policy file, by family, version, and an optional context-window preference (the `plan.model.*`, `analysis.model.*`, and `validation.model.*` properties). See Chapter 6 for the details. The set of models you can choose from is determined by the selected agent, the provider, and your credentials. Run `fortifyaa -selftest` to see it.

Advanced: custom models, gateways, and signing

If your model or endpoint is not in Pi's built-in catalog (a private deployment, a self-hosted or proxied endpoint, or a model published under another name), declare it in a `pi_models.json` file next to the executable. If reaching it needs code rather than configuration, such as an HMAC signature demanded by an API gateway, put a Pi extension in `pi_extensions/` next to the executable. Both are covered, with worked examples, in [Chapter 7](#).

3.8 Verify your setup

Before your first real scan, confirm that both credentials and model access work. The `-selftest` command verifies the setup a scan would actually use: it reads your policy (honoring `--policy`, `--config`, and `-D` overrides), determines the provider exactly as a scan would (including the auto probing described in [Section 3.7.4](#)), runs a live `report_finding` round-trip on it, and prints the model each analysis stage will use. With `--no-live` it makes no paid calls: it lists the provider candidates and stage models but does not verify connectivity.

```
fortifyaa -selftest --no-live    # list candidates / stage models, no paid call
fortifyaa -selftest             # verify the provider with a live round-trip
fortifyaa -license              # confirm your Fortify entitlement
```

Once both pass, you are ready to run a scan ([Chapter 4](#)).

4 Running a scan

This chapter describes how you actually use FAA: the recommended workflow, a basic scan, how to scope it, how to make it complementary to existing Fortify results, where the output goes, how interrupted scans resume, and how to scan only what changed. Every command-line option mentioned here is also catalogued in [Chapter 5](#).

4.1 The recommended workflow

For most teams, FAA fits into an existing Fortify SAST process as a second, deeper pass:

1. Run your normal Fortify SAST scan and obtain its results (an FPR file, a release in FoD, or an application version in SSC).
2. Run FAA locally against the same source checkout, giving it those existing results so it reports only complementary findings.
3. Upload FAA's SARIF output to your Fortify platform alongside the SAST results ([Chapter 8](#)).

You can also run FAA on its own, for example on a language that traditional SAST does not support. That is an *FAA-only* scan: simply omit the existing-results option.

4.2 Run a basic scan

The minimal command scans a project directory and writes a SARIF file:

```
fortifyaa -scan /path/to/project
```

This reads the default policy (`fortify-aa-default.properties`, next to the executable), runs the plan/analyze/validate pipeline, and writes `faa.sarif` into the current directory (add `--output-format sarif,zip` to also produce `faa.zip` for SSC

upload; Section 4.5). Token usage and estimated cost are printed as the scan progresses and summarized, per stage, at the end. If an in-flight batch produces no model activity for several minutes, FAA prints a stall warning naming the silent batch. If a scan looks hung, that warning (or its absence) tells you whether the model is still working. For a full timestamped timeline, re-run with `--debug` (Section 11.1).

4.3 Scope a scan

By default FAA considers the whole project. Use `--scope` to restrict the scan to one or more comma-separated entries, each a directory or an individual file (relative to the project directory, or absolute):

```
fortifyaa -scan /path/to/project --scope src/main/java/com/acme/auth
fortifyaa -scan /path/to/project --scope "src/auth,src/payments,config/web.xml"
```

A scoped directory is walked and filtered exactly like a full scan (file filters, git-ignore, and the plan stage all apply). An individually named file is treated as an explicit request: it is always analyzed, never dropped by a file filter or git-ignore, and never skipped by the plan stage.

Scoping reduces both cost and time, and is useful when you want to focus FAA's attention and budget on the most security-relevant parts of a large codebase.

4.4 Complementary scanning

To keep FAA's output truly complementary, give it your existing Fortify SAST results. FAA deduplicates against them downstream (deterministically and during validation), so anything SAST already reported is dropped and you see only what SAST missed. Provide existing results from exactly one source:

A local FPR file — `--fpr existing.fpr`

An FoD release — `--fod-release <id or "App[:Microservice]:Release">`, read live via fcli. This needs an fcli FoD session; use `--fcli-fod-session <name>` to select a named session.

An SSC application version — `--ssc-appversion <id or "App Name:version">`, read live via fcli. This needs an fcli SSC session; use `--fcli-ssc-session <name>` to select a named session instead of the default.

```
# Complementary to a local FPR:
fortifyaa -scan . --fpr existing.fpr --output faa.sarif

# Complementary to an FoD release (via fcli):
fortifyaa -scan . --fod-release "My App:Release 1" --output faa.sarif

# Complementary to an SSC application version (via fcli):
fortifyaa -scan . --ssc-appversion "My App:1.0" --output faa.sarif
```

The `--fpr`, `--fod-release`, and `--ssc-appversion` options are mutually exclusive: pass only one. The two live sources require fcli to be installed and an appropriate session to be logged in (Section 3.2).

4.5 Output files

SARIF is FAA's canonical output. It carries the complete result of a scan, and every other format FAA can write is derived from it: the SSC upload zip is a SARIF in a wrapper, and the Markdown reports are renderings of one. The derivation runs one way only: a Markdown report cannot be turned back into SARIF.

That has a practical consequence worth stating before the options: *keep the SARIF*. For as long as you have it you can produce any other format from it later, at no cost and without re-running anything. Discard it and the only way back to a SARIF is to pay for another scan.

It follows that there are two ways to obtain a non-SARIF format, and they produce byte-identical files:

- *Ask for it during the scan*, with `--output-format`. The scan writes every selected artifact when it finishes.
- *Convert an existing SARIF afterwards*, with `-convert` (Section 4.5.6). This makes no LLM calls and costs nothing, and works on any SARIF a previous scan produced, including one written long before you knew you wanted the other format.

Neither route is more correct than the other. Asking during the scan is convenient when you already know who will consume the results; converting afterwards is the better habit when there is more than one consumer, because it keeps one full-fidelity SARIF as the archive and treats each consumer-specific artifact as a rendering of it.

4.5.1 Choose formats

The `--output-format` option selects which artifacts are written: one value, or several comma-separated (values are case-insensitive).

sarif — the SARIF 2.1.0 result (the default). The canonical output, consumable by FoD and other SARIF-native tools.

zip — the SARIF wrapped together with a `scan.info` descriptor (`engineType=FORTIFY_AA`), ready for upload to Fortify SSC.

md-summary — a Markdown overview: one short section per finding.

md-full — a Markdown report with the full detail of every finding.

md-per-finding — a directory holding one self-contained Markdown document per finding, plus an index.

The Markdown formats are described in [Section 4.5.5](#).

4.5.2 Name the artifacts

One option names them all. `--output` gives the path of the SARIF (default `faa.sarif`), and every other artifact is named after it, with its own suffix in place of the extension:

```
fortifyaa -scan . --output out/run7.sarif \
           --output-format sarif,zip,md-summary,md-full,md-per-finding

out/run7.sarif      out/run7.zip      out/run7-summary.md
out/run7-full.md    out/run7-per-finding/
```

So you never have to guess a filename, and a single `--output` keeps a whole set of artifacts together. Directories that do not exist yet are created.

When one artifact needs to go somewhere else, override just that one with `--output-<format>`, leaving the rest named after `--output`. This is most useful for `md-per-finding`, whose directory often belongs somewhere a tool already watches:

```
fortifyaa -scan . --output-format sarif,md-per-finding \
    --output-md-per-finding .faa/issues
```

There is one such option per format: `--output-sarif`, `--output-zip`, `--output-md-summary`, `--output-md-full`, `--output-md-per-finding`. Naming a path for a format that is not in `--output-format` is an error rather than a silent no-op.

4.5.3 Keep the SARIF

Because the derived formats are one-way, a scan that writes none of the canonical formats would throw its own results away the moment it finished. FAA refuses such a run before it starts, before the license check and before any spend:

```
$ fortifyaa -scan . --output-format md-full
--output-format selects only derived formats (md-full), so this scan's results
    ↪ would be
discarded after it finishes. Markdown and FPR output cannot be converted back
    ↪ to SARIF.

Add a canonical format: --output-format md-full,sarif
Or accept the loss:     --output-format md-full --no-sarif
```

Adding `sarif` (or `zip`, which contains the SARIF) is almost always what you want. `--no-sarif` is there for the rare case where the Markdown genuinely is the only thing you need and the cost of re-scanning is acceptable; it does nothing except accept that trade-off, and the scan warns again when it finishes.

4.5.4 What the SARIF contains

Each finding is a SARIF *result*: its vulnerability category (aligned to a Fortify category), a severity together with the underlying impact and likelihood scores,

the sink location with a code snippet, a one-line summary, a human-readable description, and concrete, issue-specific remediation guidance. A stable per-finding fingerprint lets consumers track the same issue across scans. Because that description and remediation are written for the individual finding, FAA emits one SARIF rule per finding. Section 8.1 explains why, and what it means for the tools that read the file.

Beyond the findings themselves, the SARIF run metadata (the SARIF *invocation*) records how the scan was produced, so every result is traceable back to exactly how it was created:

- The effective configuration (the merged policy file plus any `-D` overrides), so the exact settings that produced the run are reproducible.
- The agent engine and the actual model resolved for each stage (plan, analysis, and validation).
- Token, cost, and time usage, both as a run total and broken down per stage.
- File-coverage counts (how many files were considered, how many were in scope after planning, and how many were analyzed), plus the full list of the analyzed files.
- Run-specific inputs, such as the complementary FPR/FoD/SSC source and any baseline or change-driven range, along with start/end timestamps and the host machine.

4.5.5 Markdown reports

Three Markdown formats render the same findings for three different readers. They carry identical content per finding: `md-summary` simply omits the long blocks, and `md-per-finding` packages the same detail one document at a time.

md-summary — a single document: a run header with the finding counts by severity, then one short section per finding giving its category, location, severity, and one-line summary. Written for a person who wants to know what the scan found, not to fix it yet.

md-full — the same document with every finding's full detail: the explanation, the remediation guidance, a short code excerpt around the sink, and the identifiers. Written to be read end to end, or searched.

md-per-finding — a directory rather than a file, holding one document per finding plus `index.md` and `index.json`.

`md-per-finding` differs from the other two in more than packaging, so it is worth being clear about when it is the right choice.

For programmatic consumption, the SARIF remains the recommended input. It is a standard format with mature libraries in most languages, it carries everything FAA produces, and it is the only output that can be converted onwards. A tool that can read SARIF should read the SARIF.

`md-per-finding` is an alternative for cases where that is impractical: a consumer that works in Markdown, a pipeline where adding a SARIF parser is not worthwhile, or a workflow that handles one finding at a time. Its structure follows from being read that way:

- Each document is self-contained, because whatever reads it sees only that one file. It opens with a YAML front-matter block carrying the finding's identity (instance id, rule id, category, severity, impact and likelihood, file and line), so a tool can route and filter without parsing prose.
- Filenames are stable across scans. A document is named after the finding's SARIF rule id, which ends in the instance id, so the same finding keeps the same filename on the next scan even if other findings come and go. A tool can therefore tell what it has already seen.
- `index.json` lists every finding with those same fields plus the name of its document, so the whole set is available in one read. Its field names mirror the SARIF, so anything it does not carry can still be looked up there by instance id or rule id. It declares a version that consumers can depend on.
- `index.md` is the same list for a person, with links to the documents.

FAA treats that directory as its own: if it already holds a previous run's output, the stale documents are removed before the new ones are written, so a finding that no longer exists does not linger. If the directory exists but holds anything FAA did not write, the run stops rather than touching it.

```
# A scan that also produces reports for people:
fortifyaa -scan . --output-format sarif,md-summary,md-full

# One document per finding, into a directory another tool reads:
```

```
fortifyaa -scan . --output-format sarif,md-per-finding \
--output-md-per-finding .faa/issues
```

4.5.6 Convert existing results

`-convert` produces any of the output formats from a SARIF you already have. It reads a `.sarif` file, or a `.zip` containing one, makes no LLM calls, and costs nothing:

```
# Reports from last month's scan:
fortifyaa -convert archive/run7.sarif --output-format md-summary,md-full

# The SSC upload artifact, from a SARIF that was scanned without it:
fortifyaa -convert archive/run7.sarif --output-format zip
```

`--output-format` is required; there is no default, since converting is an explicit request for particular artifacts. The other options work as they do for `-scan`: `--output` sets the name the artifacts derive from (by default the input's own name), `--output-<format>` overrides one of them, and `--latin1` and `--message-format` apply the conversions described below.

The input is never modified and never consumed, so there is no risk in converting the same SARIF repeatedly; FAA refuses a conversion that would write over its own input. A SARIF produced by a different tool is accepted, but FAA warns, because the reports draw on Fortify-specific content that another tool's SARIF will not carry.

4.5.7 Message format

The `--message-format` option controls how the plain-text channels of a finding are written: `message.text` (the one-line summary) and the rule texts (`fullDescription.text` with the description, `help.text` with the remediation). It affects only those: the Markdown variants and the code snippet are always emitted in their own SARIF fields regardless.

fod — the default. The plain-text channels carry their content rendered as HTML. This format exists because Fortify on Demand displays exactly

these channels, rendered as HTML, and ignores the Markdown fields; the HTML is shaped to render well there.

plain — the plain-text channels contain plain text only. Use this when the SARIF is going to a consumer that displays those channels verbatim rather than rendering them, since HTML tags would otherwise show up raw.

SSC is unaffected by this option: the SSC plugin renders the Markdown variants. So are the Markdown reports, for the same reason: they are built from the Markdown fields, which are the same whichever message format is in force.

Because the Markdown fields carry the content and the plain-text channels are derived from them, this choice is not fixed at scan time: `-convert` accepts `--message-format` too, so one archived SARIF can yield an HTML-shaped copy for FoD and a plain copy for a consumer that needs one.

```
fortifyaa -convert archive/run7.sarif --output-format sarif \
--message-format plain --output archive/run7-plain.sarif
```

4.5.8 Restricted character set (`--latin1`)

FAA's SARIF output is always UTF-8, as the SARIF standard requires. Some SSC servers, however, cannot store characters outside the Windows-1252 character set. Currently this applies to SSC running on MS SQL Server, where only the `SQL_Latin1_General_CP1_CS_AS` collation is supported. A SARIF file containing other characters (typographic symbols like $\geq$, emoji, non-Latin scripts) then fails to import.

The `--latin1` option restricts every string in the output (findings, descriptions, and code snippets alike) to characters representable in Windows-1252. It is a best-effort conversion in three layers, and never aborts the scan:

1. The LLM is instructed up front to write only Windows-1252 text.
2. Known symbols are replaced by ASCII equivalents: `>=` for $\geq$, `<=` for $\leq$, `!=` for $\neq$, `->` for $\rightarrow$, and similar sensible fallbacks.
3. Any remaining unrepresentable character becomes `?`.

The file itself remains UTF-8; the option only guarantees that every character in it exists in Windows-1252, so the import cannot fail on character-set grounds.

Use it when your SSC instance runs on MS SQL Server; leave it off otherwise, since the replacement loses information (a snippet of non-Latin source code, for example, degrades to ? characters).

-convert accepts --latin1 as well, so an existing SARIF can be restricted without re-scanning:

```
fortifyaa -convert archive/run7.sarif --output-format zip --latin1 \
--output archive/run7-ssc.sarif
```

Converting afterwards is in fact the better arrangement when the same scan serves more than one consumer: keep the full-fidelity SARIF as your archive and produce a restricted copy only for the SSC upload, rather than degrading everything at scan time. The one thing a scan-time --latin1 does that a later conversion cannot is step 1 (instructing the model to stay inside the character set in the first place), so a scan-time run needs fewer substitutions. If you know SSC on MS SQL Server is the target, pass it at scan time.

4.6 Scan state and resume

FAA scans are resumable. Every completed analysis batch and validation unit is checkpointed to a per-project build directory under the Fortify working directory (%LOCALAPPDATA%\Fortify\faa26.4 on Windows, ~/.fortify/faa26.4 elsewhere), modeled on sourceanalyzer's build session. The build id is derived from the project path; override it with -b <build-id>.

Consequences:

- A crash, an LLM error, or a budget stop never loses finished work. Simply re-running -scan auto-resumes from where it left off, skipping (and not re-paying for) completed work.
- Stages stop on the first LLM/agent error with a clear, human-readable message rather than silently degrading results. Fix the cause and re-run to resume.
- -resume -scan <dir> explicitly resumes and errors if there is nothing to resume. -clean discards the saved state; combined with -scan it forces a completely fresh scan.

```
fortifyaa -resume -scan /path/to/project # continue an interrupted scan
fortifyaa -clean -scan /path/to/project # discard state, scan fresh
fortifyaa -clean # just discard saved state
```

Cost cap across resumed runs

maxcost caps a scan, not an invocation. When a scan is resumed, the cost of the work already completed is read back from the build directory and counted towards the same cap: re-running never refills the budget. This is what makes the cap meaningful: a \$50 cap means the scan costs at most about \$50 in total, however many times you resume it.

Two consequences follow directly:

- **Resuming a capped-out scan with the same maxcost fails immediately**, before anything is spent, telling you what was spent and what the cap is. There is no budget left, so no progress is possible.
- **A raised maxcost is the new total for the whole scan**, not an extra allowance on top of what was already spent. A scan that stopped at \$50 and is resumed with -Dmaxcost=80 has \$30 of room, not \$80.

```
X Cost cap already reached for build 'a1b2c3d4': $50.1180 spent of the
$50.00 'maxcost' cap, with work still to do.
'maxcost' caps the WHOLE scan, including everything earlier invocations
already spent - resuming with the same cap has no budget to make progress.
Raise 'maxcost' (or set it to 0 for no cap) and re-run, e.g.: -Dmaxcost=200
Or run -clean to discard the saved state and start a fresh scan (which pays
again for the completed work).
```

If you would rather budget per invocation, raise the cap by the amount you are willing to spend each time you resume.

Resume with a different configuration

Saved state describes one particular scan, so FAA refuses to resume a build directory whose project, scope, existing-findings source, or policy no longer matches;

it says what differs and points at `-clean`. That protects you from silently merging results that were produced under different rules.

Two properties are exempt, because they govern how a run behaves rather than what work exists:

- `maxcost` — **raising the cap and resuming is explicitly supported.** When a scan stops at the cap, that is exactly what FAA tells you to do: raise `maxcost` (or set it to 0 for no cap) and re-run `-scan`. Completed batches are kept and not paid for again.
- `output.transcripts` — so you can turn diagnostics on to investigate a problem without discarding the state you wanted to investigate.

```
fortifyaa -scan /path/to/project -Dmaxcost=150 # resume with a higher cap
```

Anything else (a different model, effort, batch sizing, concurrency, or file-selection rule) does invalidate the saved state. Concurrency counts because `analysis.maxconcurrency` takes part in deciding how files are grouped into batches.

4.7 Keep findings stable across scans

When you scan the same project repeatedly, you want the same issue to stay the same issue. Every finding carries an *instance id*, and that is what consumers use to decide whether they have seen it before. When SSC receives an upload, a finding whose id has changed is not recognized as the one already on file: it appears as an issue that vanished plus a new one that arrived, and the audit work attached to it (the analysis, the comments, a suppression) does not follow.

FAA derives the instance id from the finding's file, its sink line, and its category. Two of those three can move while the vulnerability itself is untouched: an edit anywhere above the sink shifts its line number, and validation may settle on a slightly different, but equally defensible, category on a later run. FAA already defends against both on its own. The category is recorded only after it has been aligned to the canonical Fortify taxonomy, so wording differences do not move the id, and the sink line is anchored by matching the actual text of the line rather than trusting the line number the model reported, which drifts by a line or two between runs. That costs you nothing and needs no configuration.

--baseline <prior.sarif> addresses what remains. Given a previous scan's SARIF, FAA matches this run's findings against it and, wherever it is confident it is looking at the same finding, carries the earlier id over instead of computing a fresh one. It is confident in two situations:

- Same file and same category, with the sink within a few lines of where it was: the finding simply drifted.
- The same sink line verbatim, within a wider window, even if the category differs. The identical line of code is what establishes it is the same sink, so the id survives a category being re-aimed between runs.

Matching is one-to-one and deterministic: each baseline finding can be claimed only once, and the outcome never depends on the order findings were reported in. A finding with no confident match keeps a freshly computed id, so a baseline can only ever preserve identity; it cannot invent it. Findings that move to a different file between runs are deliberately not reconciled, as establishing that reliably would need dataflow analysis that FAA does not perform.

This is not limited to change-driven scans. A plain full scan accepts --baseline and benefits from it in exactly the same way, which is the usual case when you re-scan a project periodically:

```
fortifyaa -scan . --baseline previous.sarif --output current.sarif
```

Point it at the most recent scan of the same project. The baseline is only read, never written to, so keeping the previous SARIF alongside the new one is all the bookkeeping required. The scan reports how many ids it carried over, so you can see whether the baseline did any work. A baseline that cannot be read does not fail the scan: FAA says so and continues with freshly computed ids, rather than discarding analysis you have already paid for.

The change-driven modes described next build on the same mechanism: --incremental requires a baseline because its output is the updated baseline, while --differential merely accepts one.

4.8 Change-driven scans: incremental and differential

Because a full FAA scan has a real cost, you often do not want to re-analyze an unchanged codebase. Two change-driven modes scope the (paid) analysis to the files a change actually touched. Both require a git repository. They differ in what the output contains, and in whether a prior scan (`--baseline <prior.sarif>`) is needed, as the following table shows.

Mode	Scans	Output	Baseline
<code>--incremental <range></code>	changed files	a <i>complete</i> SARIF: the baseline, updated for the changed files (findings for untouched files are carried over verbatim)	required
<code>--differential <range></code>	changed files	<i>only the issues introduced</i> on changed lines	optional

The `<range>` is whatever `git diff` accepts, so there is no new syntax to learn. The following table shows the common cases.

You want	<code><range></code>
A PR / branch (what it introduced)	<code>base...head</code> (three-dot = merge-base..head)
An explicit commit range	<code>from..to</code>
One commit's own changes	<code><sha>~1..<sha></code>
Uncommitted working-tree edits	<code>HEAD</code>

```
# A PR's worth of changes, full updated report:
fortifyaa -scan . --baseline prev.sarif --incremental main...HEAD --output
```

→ updated.sarif

```
# Only what this PR introduced (useful for gating) -- no baseline needed:
fortifyaa -scan . --differential main...HEAD --output introduced.sarif
```

For `--differential`, “introduced” means a finding whose sink line falls on a line the change added or modified. This is deterministic and immune to scan-to-scan recall jitter: it asks “is this on new code?”, not “was this absent from a prior scan?”. Because that judgment comes from the change itself, no baseline is required, which makes `--differential` the natural fit for PR decoration when a full baseline scan of the codebase is not cost-feasible. A baseline is still accepted: when given (ideally a scan of the change’s base), findings that were already present keep their instance ids under the matching described in Section 4.7, even if the change moved them to a different line.

Limitation: cross-file effects. Both modes analyze only the changed files. A change in one file that creates a vulnerability whose sink is in an unchanged file will not be re-derived, because that file is not scanned. A full baseline scan (no mode flag) remains the way to catch cross-file drift.

4.9 Manage cost during a scan

Every scan spends real money on LLM calls. The policy file’s `maxcost` property sets a soft USD cap, and the choice of model per stage and which stages run all affect cost. Cost management is important enough that it has its own treatment in Chapter 10, with the relevant properties documented in Chapter 6.

5 Command reference

This chapter documents every FAA command and option. Following the Fortify sourceanalyzer convention, verbs use a single dash (`-scan`, `-clean`, ...) while options use a double dash (`--output`, `--scope`, ...). The single exceptions are the short forms `-o` (for `--output`), `-b` (for `--build-id`), and the `-D` property override.

5.1 Commands (verbs)

The following table lists the commands.

Command	Purpose
<code>-scan <projectDir></code>	Scan a project for security issues and write SARIF. Resumes automatically if a prior run was interrupted.
<code>-convert <file></code>	Produce the other output formats from a SARIF you already have (a <code>.sarif</code> , or a <code>.zip</code> containing one). Makes no LLM calls and costs nothing; the input is never modified. See Section 4.5.6 .
<code>-clean [-scan ...]</code>	Discard saved scan state for this project. Used alone it just cleans; combined with <code>-scan</code> it forces a fresh scan.
<code>-resume -scan <dir></code>	Resume an interrupted scan; errors if there is nothing to resume.
<code>-selftest</code>	Verify the policy's provider and models with a live report_finding round-trip (with <code>pi.provider=auto</code> it probes the candidates and reports which provider gets used).
<code>-license [clear]</code>	Verify Fortify access (FoD/SSC); <code>clear</code> resets the 24-hour cache.
<code>-pilogin</code>	Interactively store LLM credentials: a subscription sign-in or an API key (Section 3.7.2).

Continued on next page

Command	Purpose
-help	Show usage.

5.2 -scan options

These are the run-specific inputs and outputs. Everything that tunes how a scan behaves lives in the policy file instead (Chapter 6).

Option	Default	Purpose
--output <file>, -o	faa.sarif	Output path for the SARIF, and the name every other selected artifact derives from (<name>.zip, <name>-summary.md, and so on). Missing directories are created. See Section 4.5.
--output-format <formats>	sarif	Which artifacts to write: sarif, zip (the SARIF wrapped for SSC upload), md-summary, md-full, md-per-finding (a directory). Comma-separated, case-insensitive. See Section 4.5.
--no-sarif	off	Accept that this scan keeps no canonical output. Only needed when --output-format selects neither sarif nor zip, which FAA otherwise refuses, because the results could not be converted afterwards.

Continued on next page

Option	Default	Purpose
<code>--output-<format> <path></code>	derived from <code>--output</code>	Put one format's output at an exact path, leaving the others named after <code>--output</code> ; for example <code>--output-md-per-finding .faa/issues</code> . One per format name above. Naming a format that is not being written is an error.
<code>--message-format <fmt></code>	fod	How the plain-text channels (<code>message</code> , <code>text</code> , <code>rule texts</code>) are written: <code>fod</code> shapes them as HTML for Fortify on Demand; <code>plain</code> emits plain text only. Does not affect the Markdown reports. See Section 4.5.7.
<code>--latin1</code>	off	Restrict every string in the output to the Windows-1252 character set, for SSC servers whose database cannot store anything else (SSC on MS SQL Server). Best effort: known symbols get ASCII substitutes (for example, <code>></code> <code>=</code>), any remaining character becomes <code>?</code> . See Section 4.5.8.
<code>--scope <path[,path...]></code>	whole project	Restrict the scan to one or more comma-separated directories and/or files (relative to the project, or absolute). Directories are walked and filtered as usual; an individually named file is always analyzed (it bypasses the file filters and the plan stage).
<code>--fpr <file></code>	none	Existing Fortify results from a local FPR; report complementary findings only.

Continued on next page

Option	Default	Purpose
<code>--fod-release <rel></code>	none	Existing results from an FoD release (id, or "App[:Microservice]:Release"), read live via fcli.
<code>--fcli-fod-session <name></code>	default session	Use a named fcli FoD session (only with <code>--fod-release</code>).
<code>--ssc-appversion <av></code>	none	Existing results from an SSC application version (id, or "App Name:version"), read live via fcli.
<code>--fcli-ssc-session <name></code>	default session	Use a named fcli SSC session (only with <code>--ssc-appversion</code>).
<code>--baseline <file></code>	none	A prior FAA SARIF whose instance ids are reused for findings that match across line/category drift, so a re-scan keeps the same issue identity (Section 4.7). Valid in any scan, including a full one; the change-driven modes additionally build on it.
<code>--incremental <range></code>	off	Re-scan only changed files; emit a complete SARIF updated from the baseline. Requires <code>--baseline</code> and a git repo.
<code>--differential <range></code>	off	Emit only issues introduced on changed lines. Requires a git repo; <code>--baseline</code> is optional (adds instance-id continuity with a prior scan).
<code>--policy <name></code>	default	Load <code>fortify-aa-<name>.properties</code> from the executable's directory.
<code>--config <path></code>	—	Load an explicit <code>.properties</code> file (absolute or cwd-relative).

Continued on next page

Option	Default	Purpose
-Dkey=value	—	Override a single policy property (repeatable), for example -Dplan.enabled=false.
-b <build-id>, --build-id	derived from path	Override the scan-state build id (Section 4.6).
--keep-scratch	off	Keep the temporary scratch directory for debugging.
--log <path>	log/faa.log	Write the run log to <path> instead of the default log/faa.log under the Fortify working directory (the directory is created if needed), and mirror into that same file everything the scan prints to the console (phase banners, findings, checkpoints, the final cost and timing summary), so the file is a complete record of the run. Without this option the run log holds only the timestamped halt and error entries, in the shared faa.log.
--pi <path>	pi_models.json	Load the custom model catalog (Chapter 7) from an explicit file, instead of pi_models.json next to the executable.

Continued on next page

Option	Default	Purpose
<code>--debug</code>	off	Write a timestamped diagnostic timeline (per-batch dispatch, agent tool calls, LLM request/response lines, completion, cost, budget decisions, stall warnings) to <code>debug.log</code> in the scan-state build directory, and persist the full agent transcripts alongside it. When an LLM call fails, the exact request payload sent to the endpoint and the HTTP response received are additionally dumped to <code>lm-failures/<unit>.json</code> in the same directory. Use this when a scan appears to hang or misbehave: the log shows what every batch was last doing, and when, and for a failing provider, precisely what was on the wire (Section 11.1).

Mutual exclusions. `--fpr`, `--fod-release`, and `--ssc-appversion` are mutually exclusive: pass at most one source of existing findings. `--incremental` and `--differential` are mutually exclusive; only `--incremental` requires `--baseline`.

5.3 -convert options

`-convert <file>` reads a SARIF (a `.sarif`, or a `.zip` containing one) and writes the requested output formats from it. It needs no policy, no credentials, and no license check, and it never modifies its input. See Section [4.5.6](#).

Option	Default	Purpose
<code>--output-format <formats></code>	—	<i>Required.</i> The artifacts to write, from the same list as <code>-scan</code> . There is no default: converting is an explicit request for particular artifacts.
<code>--output <file>, -o</code>	named after the input	The name the artifacts derive from, exactly as for <code>-scan</code> . By default the input's own name is used, so <code>run7.sarif</code> yields <code>run7-summary.md</code> .
<code>--output-<format> <path></code>	derived from <code>-output</code>	Put one format's output at an exact path, as for <code>-scan</code> .
<code>--message-format <fmt></code>	unchanged	Re-shape the SARIF's plain-text channels (Section 4.5.7). Omit to keep them as the scan wrote them.
<code>--latin1</code>	off	Restrict all output text to Windows-1252 (Section 4.5.8).

FAA refuses a conversion that would overwrite its own input, and warns when the input's SARIF was produced by a tool other than FAA (the reports draw on Fortify-specific content that another tool's SARIF will not carry).

5.4 Selecting and overriding configuration

The policy that drives a scan is located as follows:

Form	Resolves to
(default)	<code>fortify-aa-default.properties</code> , next to the executable
<code>--policy <name></code>	<code>fortify-aa-<name>.properties</code> , next to the executable
<code>--config <path></code>	the explicit file at <code><path></code>
<code>-Dkey=value</code>	overrides a single property after the file is loaded (last one wins)

5.5 -selftest options

-selftest reads the same policy a scan would and verifies exactly what the scan uses. Select the policy with the same --policy, --config, and -D options as for -scan. The following table lists the options.

Option	Purpose
--no-live	Make no paid calls: list the provider candidates and the per-stage models, but do not verify connectivity.
--agent <pi claude-code fake>	Override the agent backend to test (fake needs no policy or credentials).
--model <id>	Test a specific raw model id (diagnostic; bypasses the policy's model selection).
--policy <name>, --config <path>, -Dkey=value	Select or override the policy, exactly as for -scan.

5.6 -license options

-license takes one optional argument:

Argument	Purpose
(none)	Verify Fortify access, using the 24-hour cache.
clear	Reset the cached license result before checking.

5.7 -pilogin options

-pilogin accepts the following arguments:

Argument	Purpose
(none)	Interactive login: choose subscription or API key, then the provider.
--provider <id>	Skip the provider menu (for example --provider anthropic).
status	List stored credentials (provider, kind, and token expiry), never the secret itself.
clear [<provider>]	Remove one stored credential, or all of them when no provider is given.

Stored credentials take precedence over environment variables for that provider. For details and the storage location, see Section [3.7.2](#).

5.8 Where scan state is saved

Scan state is saved under the Fortify working directory, like sourceanalyzer's build session: %LOCALAPPDATA%\Fortify\faa26.4 on Windows, and ~/.fortify/faa26.4 elsewhere. Each scan has its own build directory there, build/<build-id>. The build id is derived from the project path; override it with -b <build-id>. For how resume works, see Section [4.6](#).

6 Configuration: the policy file

Everything that tunes how a scan behaves (the agent, the models, which stages run, effort, batch sizing, concurrency, and the cost cap) lives in an external `.properties` policy file, never built into the executable. A bare `-scan <dir>` is driven entirely by `fortify-aa-default.properties`, shipped next to the binary. Select another shipped or custom policy with `--policy <name>`, point at any file with `--config <path>`, and override individual properties with `-Dkey=value` (Chapter 5).

Every property is required. There are no built-in defaults: a property that is absent from both the policy file and any `-D` override (even one belonging to a disabled stage) is a startup error. The whole configuration is validated up front, before the license check or any LLM spend, and all problems are reported at once, so you never pay for analysis only to hit a configuration error later. The effective, merged configuration is embedded in the output SARIF for traceability.

The remainder of this chapter documents every property, grouped by topic. The “Default” column shows the value shipped in `fortify-aa-default.properties`, which is tuned for a deep, complementary scan.

6.1 Global settings

These properties apply to the scan as a whole.

Property	Default	Meaning
agent	pi	The coding-agent backend: <code>pi</code> (the bundled, in-process agent) or <code>claude</code> (your separately installed Claude Code CLI). A fake backend exists for no-cost plumbing tests.

Continued on next page

Property	Default	Meaning
<code>pi.provider</code>	auto	Pi agent only (ignored when agent=claude): selects which provider serves the chosen model. auto probes the candidate providers at scan start with one tiny live call (providers you signed into with <code>-piloin</code> first) and uses the first that answers, fixed for the whole scan. Pin a concrete id, such as <code>amazon-bedrock</code> , <code>google-vertex</code> , <code>azure-openai-responses</code> , <code>anthropic</code> , <code>github-copilot</code> , or <code>openai</code> (see the ids listed by <code>-selftest</code>), to make runs fully explicit; a pinned provider is still probed but never substituted. See Section 3.7.4.
<code>maxcost</code>	50	Maximum USD to spend on a scan, not on one invocation: a resumed scan continues the same running total (Section 4.6). The first analysis batch runs alone to calibrate the per-batch cost; after that, batches are only dispatched while the projected total cost, including work in flight, stays within the cap, and a slice of the cap is held back so validation can review what analysis found (Chapter 10). Work already in flight runs to completion, so treat it as a close approximation rather than a hard guarantee; 0 disables the cap. Raise it before scanning large codebases.
<code>output.transcripts</code>	false	Persist the full agent conversation of every run under the build directory's <code>transcripts/</code> folder, for debugging or read-behavior analysis. Off by default: transcripts embed the source the agent read, so they add up on disk. <code>--debug</code> turns this on automatically.

6.2 File enumeration

Before any LLM work, FAA always walks the scope deterministically and drops files not worth analyzing. These properties define what gets dropped. This walk runs whether or not the plan stage is enabled: “plan disabled” means every enumerated source file is analyzed, not that this basic filtering is bypassed.

Property	Meaning
<code>files.excludedirs</code>	Comma-separated directory names skipped wholesale, matched by exact name at any depth (a value may span lines with a trailing backslash). Default covers VCS, dependency, build-output, cache, and IDE directories (<code>.git</code> , <code>node_modules</code> , <code>target</code> , <code>build</code> , <code>dist</code> , <code>.idea</code> , ...).
<code>files.excludeextensions</code>	Comma-separated file extensions (no leading dot, case-insensitive) treated as non-source. Default covers images, fonts, media, archives, compiled binaries, office documents, and data/lock/map files.
<code>files.excludenames</code>	Comma-separated glob patterns (<code>*</code> , <code>?</code>), matched case-insensitively against each file’s name. Default covers minified JS/CSS, common lockfiles, the Maven/Gradle wrapper launchers, and <code>.git</code> gitlink files.
<code>files.maxbytes</code>	Skip files larger than this many bytes (oversized or generated blobs). Default 1000000. Empty files are always skipped.
<code>files.excludegitignored</code>	true/false: also skip anything git ignores. Default true. When true, git must be on PATH and the project should be a git checkout; if git cannot be run the scan aborts (set this to false for non-repo trees).

6.3 Per-stage model selection and effort

Each of the three stages selects its model and thinking effort independently, with the same family of properties (`<stage>` is `plan`, `analysis`, or `validation`).

Property	Meaning
<code><stage>.effort</code>	The LLM thinking level for this stage only: <code>low</code> , <code>medium</code> , <code>high</code> , or <code>max</code> . It affects reasoning depth, not batching or concurrency.
<code><stage>.model.family</code>	The model family, for example <code>opus</code> , <code>sonnet</code> , <code>haiku</code> , or <code>fable</code> . The available families depend on the selected agent and provider. Matched as part of the model id and without a provider prefix (<code>gpt-5.1</code> , not <code>openai/gpt-5.1</code> ; the provider comes from <code>pi.provider</code>). A value equal to a whole model id selects exactly that model, so <code>gpt-5.1</code> means <code>gpt-5.1</code> and not <code>gpt-5.1-codex</code> .
<code><stage>.model.version</code>	<code>latest</code> , a <code>major.minor</code> (for example, <code>4.6</code>), or a bare <code>major</code> for single-number families (for example, <code>5</code>).
<code><stage>.model.context</code>	Context-window preference: <code>default</code> , <code>max</code> , <code>1m</code> , or <code>200k</code> (honored by the <code>Pi/Bedrock</code> path).

The default policy matches each stage’s model to its job: a cheap, fast model for the mechanical plan pass (`haiku`), a cost-effective one for the heavy analysis stage (`sonnet`, which matches stronger models on the stable core of findings at a fraction of the cost), and a strong one for the judgment-heavy validation stage (`opus`). Analysis runs at `effort=high`, because it carries the recall-critical reasoning, while plan and validation run at `medium`. Adjust these to trade cost against depth.

6.4 Plan stage

An optional, LLM-driven pass that prioritizes the enumerated files and skips low-value ones. It does not perform the basic filtering above (that always runs first).

Property	Default	Meaning
plan.enabled	true	Run the plan stage. When false, every enumerated source file is analyzed.
plan.effort	medium	Thinking level (see Section 6.3).
plan.batchsize	25	Files classified per plan turn. Larger means fewer, cheaper turns; smaller means finer-grained ones.
plan.maxconcurrency	16	Upper bound on plan chunks classified in parallel (clamped to the number of chunks).
plan.model.family	haiku	Model family for the plan stage.
plan.model.version	latest	Model version.
plan.model.context	default	Context-window preference.

6.5 Analysis stage

The stage that actually looks for vulnerabilities. Coverage is a property of the driver, not the agent: FAA enumerates the scope into batches and analyzes them with bounded concurrency.

Property	Default	Meaning
analysis.enabled	true	Run the analysis stage. Normally always on; turning it off while plan is on yields a plan-only diagnostic that writes no SARIF.
analysis.effort	high	Thinking level. Set to high by default: this stage carries the recall-critical reasoning.
analysis.model.family	sonnet	Model family for analysis.
analysis.model.version	latest	Model version.
analysis.model.context	default	Context-window preference.

Continued on next page

Property	Default	Meaning
analysis.minbatchsize	3	Smallest derived batch (small repos parallelize more).
analysis.maxbatchsize	30	Largest derived batch (large repos stay bounded).
analysis.maxconcurrency	16	Upper bound on batches analyzed in parallel.
analysis.samples	1	Number of independent analysis passes over the scope (the union is taken).

How batch size is derived. FAA does not use a fixed batch size. It derives the files per batch from the scope so small repositories parallelize and large ones stay bounded:

$$\text{filesPerBatch} = \left\lceil \frac{\text{filesInScope}}{\text{maxconcurrency}} \right\rceil, \text{ clamped to } [\text{minbatchsize}, \text{maxbatchsize}].$$

Smaller batches give the agent less latitude per turn (more consistent recall) but cost more turns; larger batches are cheaper and preserve more cross-file context.

About analysis.samples. The agent samples at a non-zero temperature, so a borderline finding may appear in one pass and be missed in another. Running the analysis N times and unioning the findings reduces the chance of missing such a finding to roughly $(1 - p)^N$ for a finding seen with per-pass probability p . This costs about N times the analysis stage (validation deduplicates the union and is unaffected). Leave it at 1 for cheap exploratory scans; raise it to 2 or 3 for the infrequent, high-value runs where reproducibility matters most.

6.6 Validation stage

Reviews the analysis output to drop false positives, deduplicate, and align categories to the Fortify taxonomy. It reviews analysis output, so it runs only when `analysis.enabled=true`.

Property	Default	Meaning
validation.enabled	true	Run the validation stage. Turning it off saves cost but degrades output quality.
validation.effort	medium	Thinking level.
validation.model.family	opus	Model family for validation (set to sonnet to cheapen it).
validation.model.version	latest	Model version.
validation.model.context	default	Context-window preference.
validation.maxbucketsize	40	Findings reviewed together in one holistic turn. Above this, a cheap pass clusters likely duplicates and each cluster is reviewed in parallel, bounded to this many findings per turn.
validation.maxconcurrency	8	Upper bound on validation units reviewed in parallel (clamped to the number of units).

6.7 Example: a budget-capped complementary scan

Overrides can be combined with a policy on the command line. For instance, to run a large repository complementary to an existing FPR, with a tighter budget and a cheaper validation model:

```
fortifyaa -scan /path/to/project \
  --fpr existing.fpr --output faa.sarif \
  -Dmaxcost=150 -Dvalidation.model.family=sonnet
```

7 Extending Fortify Agentic Analyzer for your environment

FAA talks to LLMs through its bundled Pi agent, which supports a wide range of providers by default (Chapter 3). Real enterprise deployments are not always in that range: the model may sit behind an API gateway that demands a signature scheme of its own, or be published under a private name, or be reached through layers of proxies. FAA cannot support every such arrangement natively, but it does not have to. Two files next to the `fortifyaa` executable let you describe your environment, as the following table summarizes.

File	Governs	Use it when
<code>pi_models.json</code>	What models exist and where they live: providers, endpoints, model ids, wire protocol, capabilities.	Your model or endpoint is not in the built-in catalog: a private deployment, a self-hosted or proxied endpoint, or a renamed model.
<code>pi_extensions/</code>	How requests are made: code that runs inside the scan and can rewrite outgoing headers and payloads.	Reaching the endpoint needs logic, not just configuration: gateway signatures, token exchange, or request rewriting.

Both are optional, both are read from the executable's own directory (the same place as your policy files), and neither requires a stand-alone Pi installation. FAA deliberately does not read Pi's own locations: not `~/.pi/agent/models.json`, not `~/.pi/agent/extensions/`, and not a `.pi/` directory inside the project being scanned. A scan is defined entirely by the FAA install that runs it, so it behaves

the same on a developer laptop and a build agent, and scanning a repository never executes code that repository ships.

7.1 Custom models and providers (pi_models.json)

Place a `pi_models.json` file next to the `fortifyaa` executable. Its format is Pi's models-configuration schema: a top-level "providers" object. The full schema, with every supported field, is documented by Pi at <https://pi.dev/docs/latest/models>; this section covers only what matters for FAA.

A minimal example, an OpenAI-compatible endpoint published through a company gateway:

```
{
  "providers": {
    "acme-gateway": {
      "baseUrl": "https://llm.acme.example/openai/v1",
      "api": "openai-responses",
      "apiKey": "$ACME_GATEWAY_TOKEN",
      "models": [
        {
          "id": "gpt-5.1",
          "name": "GPT-5.1 (via Acme gateway)",
          "reasoning": true,
          "input": ["text"],
          "contextWindow": 400000,
          "maxTokens": 128000,
          "cost": { "input": 1.25, "output": 10, "cacheRead": 0.125, "
↪ cacheWrite": 0 }
        }
      ]
    }
  }
}
```

When the file is present its definitions are merged over the built-in catalog; when it is absent FAA uses the built-in catalog only. Reference the provider from your policy as usual: `pi.provider=acme-gateway` plus `*.model.family` values that match

the model ids you declared. `fortifyaa -selftest` lists what FAA ended up with, and a malformed `pi_models.json` is a hard error rather than a silent fallback.

7.1.1 Point a built-in provider at your endpoint

The example above declares a provider from scratch, which means declaring its protocol and capabilities too. When your gateway fronts a provider FAA already knows (OpenAI, Azure OpenAI, Anthropic, or Bedrock), there is a much better option: name the built-in provider id and override only the endpoint.

```
{
  "providers": {
    "openai": { "baseUrl": "https://llm.acme.example/openai/v1" }
  }
}
```

That is the whole file. Everything else about the provider is inherited: all of its model entries, each with the wire protocol, thinking-level mapping, context limits and compatibility flags that upstream maintains and tests. Your policy keeps naming models the normal way (`analysis.model.family=gpt-5.1`), and requests go to your gateway instead of the vendor.

Prefer this form whenever it applies. Re-declaring a known model under a new provider means taking ownership of fields whose correct values are not obvious, and getting them wrong misbehaves rather than fails (Section [7.1.2](#)).

Where the endpoint is configured, and where it is not

The endpoint is provider configuration, so it belongs in `pi_models.json`. Extensions cannot introduce an endpoint; they act on requests that are already addressed. A gateway integration therefore normally uses both files, each for its own half of the problem:

File	Responsibility
pi_models.json	Where requests go, and what the model is: the gateway baseUrl, and either an override of a built-in provider (Section 7.1.1) or a full declaration for a model FAA does not know.
pi_extensions/	What each request carries: a computed signature, a fetched token, or a rewritten payload; anything that has to be produced per request rather than configured once.

Both worked examples below assume the endpoint has already been set this way: the signing extension in Section 7.2.1 signs requests that are already going to the gateway, and the rename in Section 7.2.2 adjusts the model name in a request already addressed to it.

7.1.2 A model entry is a behavioral contract, not a label

This is the single most important thing to understand before declaring a model.

A catalog entry does not merely give a model a name to send to an endpoint. It tells Pi how to talk to that model: which wire protocol to use (api; openai-responses and openai-completions are different protocols, with different shapes for tool calls and tool results), whether the model reasons and how thinking levels map onto its parameters, its context and output limits, and, through compat, a set of per-model quirks: whether the server understands the developer role, whether it accepts reasoning_effort, whether tool results must carry the tool name, and more.

The practical consequence: **a model reached under a different name is not automatically the model it resembles**. If a gateway republishes GPT-5.1 as acme-gpt51, declaring it with the wrong api or without the compat flags its backend needs will not produce a clean error; it produces misbehavior. The most common symptom is a tool-calling loop: the model calls a tool such as report_plan, the result is threaded back in a shape it does not recognize, and it calls the same tool again, indefinitely. If you see a stage repeating one tool call, suspect the model declaration before anything else, and compare your entry against the built-in entry for the model it is based on.

7.1.3 Is the Pi model catalog exhaustive?

Pi's model catalog (<https://pi.dev/models>) is not a restriction on what FAA can run: `pi_models.json` exists precisely so you can go beyond it. But it is not merely a convenience list either: it is the set of models whose protocol and quirks have been calibrated and are maintained upstream.

Treat it as the supported set:

- **A model in the catalog**, reached through its built-in provider, is known to work. Report a problem and it can be fixed upstream.
- **A model outside the catalog**, including a catalog model republished under another name, is configurable, not automatically supported. The `api`, `compat`, and `capability` fields you declare become your responsibility, and getting them wrong shows up as odd behavior rather than as a clear failure (Section 7.1.2).

When a gateway renames a model, the most robust arrangement is to keep the name Pi knows and let an extension rewrite it on the way out (Section 7.2.2).

7.2 Extensions (`pi_extensions/`)

Some environments cannot be described by configuration alone. If every request must carry an HMAC signature computed over its body, or a short-lived token has to be fetched and attached, or the payload needs reshaping for a proxy, you need code in the request path. Pi's extension mechanism provides exactly that, and FAA exposes it.

Create a `pi_extensions` directory next to the `fortifyaa` executable and put TypeScript modules in it:

```
<install dir>/
  fortifyaa(.exe)
  fortify-aa-default.properties
  pi_models.json          (optional)
  pi_extensions/
    gateway-auth.ts       loaded
```

rewrite/index.ts	loaded (subdirectory with an index)
notes.md	ignored

Every .ts file directly in `pi_extensions/`, plus an `index.ts` inside any subdirectory, is loaded in alphabetical order at the start of every agent session. No configuration property is needed: presence is enough. If a file fails to load, the scan stops with an error naming it, because an extension that silently did not load would mean requests going out unsigned.

7.2.1 What an extension looks like

An extension is a module whose default export receives Pi's `ExtensionAPI` and subscribes to events. The complete API (every event, its payload, and the context object) is documented by Pi at <https://pi.dev/docs/latest/extensions>. The two events that matter most for gateway integration are:

Event	What you can do
<code>before_provider_headers</code>	Mutate <code>event.headers</code> in place before the request is sent: add or replace a header (set a string), or remove one (set <code>null</code>).
<code>before_provider_request</code>	Inspect the assembled provider payload, and optionally return a replacement object to send instead.

A signing extension, in the shape an API gateway might expect:

```
// pi_extensions/gateway-auth.ts
import { createHmac } from "node:crypto";
import type { ExtensionAPI } from "@earendil-works/pi-coding-agent";

export default function (pi: ExtensionAPI) {
  const keyId = process.env.GATEWAY_KEY_ID ?? "";
  const secret = process.env.GATEWAY_SECRET ?? "";

  pi.on("before_provider_headers", (event) => {
    const date = new Date().toUTCString();
    const signature = createHmac("sha256", secret)
```

```
.update(`date: ${date}`)
.digest("base64");

event.headers["date"] = date;
event.headers["authorization"] =
  `hmac username="${keyId}", algorithm="hmac-sha256", ` +
  `headers="date", signature="${signature}"`;
});
}
```

Keep secrets in the environment, not in the extension file. The exact string to sign (which headers, in which order, and whether the body digest is included) is defined by your gateway's configuration; consult it rather than copying the example verbatim.

This extension is only half of the setup: it says what every request carries, not where requests go. The endpoint comes from a `pi_models.json` alongside it, ideally as a one-line override of the built-in provider your gateway fronts:

```
{
  "providers": {
    "azure-openai-responses": { "baseUrl": "https://gateway.acme.example/azure/
    ↪ v1" }
  }
}
```

7.2.2 Rename a model on the way out

When a gateway publishes a known model under a private name, declaring that name as a new model means re-declaring its protocol and quirks, with the risks described in Section 7.1.2. The alternative is to let FAA and Pi use the model they already know, and rename it in the last step before the request leaves:

```
// pi_extensions/model-name.ts
import type { ExtensionAPI } from "@earendil-works/pi-coding-agent";

const OUTBOUND: Record<string, string> = { "gpt-5.1": "acme-gpt51" };
```

```
export default function (pi: ExtensionAPI) {
  pi.on("before_provider_request", (event) => {
    const payload = event.payload as { model?: string };
    const renamed = payload.model ? OUTBOUND[payload.model] : undefined;
    return renamed ? { ...payload, model: renamed } : undefined;
  });
}
```

Your policy keeps saying gpt-5.1, so Pi keeps using the calibrated protocol, thinking levels, and quirks for that model; only the name on the wire changes. Note that this is a rename, not a translation: it is correct only when the gateway really does serve the same model behind that name.

As with signing, the endpoint is configured separately: the accompanying `pi_models.json` redirects the built-in provider to the gateway:

```
{
  "providers": {
    "openai": { "baseUrl": "https://gateway.acme.example/v1" }
  }
}
```

Together, the two files say: talk to the gateway, using everything FAA knows about gpt-5.1, but call it acme-gpt51 on the wire. Compare that with declaring acme-gpt51 as a new model, which would require restating its protocol and quirks correctly, and would fail quietly if you did not.

7.2.3 Develop and test an extension

Extensions load at the start of every agent session, so the fast way to work on one is the self-test rather than a scan:

```
fortifyaa -selftest --no-live      # free: loads extensions, lists models, no
    ↪ LLM call
fortifyaa -selftest                # adds one real round-trip through your
    ↪ gateway
```

--no-live makes no paid calls at all and still loads every extension, so it is the loop to iterate in: edit, run, read the error, repeat. It reports what it loaded, which is also the quickest way to confirm a file is in the right place:

```
Ext:      1 loaded
          /opt/fortifyaa/pi_extensions/gateway-auth.ts
```

Once it loads cleanly, drop --no-live to make one real request; that is what proves the gateway accepts what your extension produces. Only then is a scan worth starting.

When an extension fails to load

A file that cannot be loaded stops the run (at the self-test, and before any spend in a scan), naming the file, the reason, and the position:

```
X Failed to load 1 Pi extension(s) from /opt/fortifyaa/pi_extensions:
  /opt/fortifyaa/pi_extensions/gateway-auth.ts: Failed to load extension:
  ParseError: Unexpected token, expected ",", gateway-auth.ts:7:0
  Fix or remove the file; FAA will not run with an extension that failed to load.
```

This is deliberately fatal rather than a warning: an extension that quietly failed to load would mean requests going out unsigned, unrewritten, or otherwise not as intended. The same treatment covers a module that imports something that cannot be resolved (Cannot find package '...' from ...) and a factory that throws while initializing; its own error message is reported as-is.

Type errors are not checked

Extensions are transpiled, not type-checked: type annotations are erased at load time and never verified. A file with a genuine type error (assigning a string to a number, or calling an event that does not exist) **loads without complaint** and misbehaves later, if at all. Only syntax errors and load-time exceptions are caught.

If you want type checking (recommended for anything beyond a few lines), develop the extension in a small scratch project with the Pi agent package installed as a dev dependency, and run tsc or use an editor with TypeScript support:

```
npm install --save-dev @earendil-works/pi-coding-agent typescript
npx tsc --noEmit gateway-auth.ts
```

Then copy the file into `pi_extensions/`. The package is needed only for types: extensions import it with `import type`, which disappears at load time, so nothing has to be installed next to the deployed extension.

Printing from an extension

`console.error(...)` from inside an extension appears in FAA's output, which is the simplest way to see whether a hook ran and what it saw. Combine it with `--debug` on a scan: `debug.log` carries the request/response timeline your hook sits in, and `llm-failures/` holds the exact payload of anything that failed. Do not log secrets, including signatures and tokens.

7.2.4 Limits and responsibilities

- **Extensions are code you run.** They execute with the permissions of the FAA process, inside a scan that holds LLM credentials and reads your source. Treat an extension exactly like any other code you deploy: review it, keep it in version control, and install it deliberately.
- **Tools registered by an extension are not exposed to the model.** FAA gives each session an explicit tool allow list (Chapter 9), so an extension shapes transport and environment, not what the agent is able to do. This is deliberate and not configurable.
- **Pi's extension API is Pi's.** FAA decides which files load; everything inside them is the API documented at <https://pi.dev/docs/latest/extensions>. Extensions written for stand-alone Pi generally work unchanged, as long as they do not depend on interactive-mode features (there is no TUI in a scan).

7.3 Diagnose a custom setup

Run `fortifyaa -selftest` first: it exercises credentials, provider selection, and a real round-trip through your gateway and extensions, for a fraction of a cent, and reports the model each stage will use.

When a scan itself misbehaves, add `--debug`. It writes into the scan's build directory (the path is printed at startup):

Artifact	Contents
<code>debug.log</code>	A timestamped timeline per unit: dispatch, every tool call, LLM request and response lines, warnings including rejected tool payloads, stall warnings, and completion with status.
<code>llm-failures/</code>	For any run that did not complete: the exact request payload as it went on the wire, plus the HTTP response.
<code>transcripts/</code>	The full conversation for every run. <code>--debug</code> turns these on automatically. A run still in flight has a <code>.partial.json</code> file that is refreshed as it goes, so a scan you interrupt, or one that hangs, still leaves the conversation up to that moment.

A repeating tool call in `debug.log` accompanied by `Rejected invalid ... warnings` means the model's payloads are not matching the expected schema; the same pattern without rejections means the model is not seeing its tool results, which points at the model declaration or the gateway rather than at the model itself (Section 7.1.2). If you need help from OpenText, see Chapter 11 for what to send.

Runaway tool calls stop themselves

Nothing in the LLM protocol limits how often a model may call a tool while working on one request, so a model that cannot satisfy a tool's schema (or that never sees its tool results) would otherwise retry indefinitely, and every attempt is

billed.

FAA therefore ends a run after **three rejected payloads in a row** for the same tool, and reports it like any other agent failure:

```
[batch-3] stopped after 3 invalid finding payloads in a row (tool schema);
      last error: /line: Expected number
```

The scan halts with that cause and stays resumable: the affected unit is not checkpointed, so once the model or its declaration is fixed, -scan picks up from there without re-paying for the work already done. The counter is consecutive, and any accepted payload resets it: a model that reports ten good findings and fumbles one is working normally, not looping.

8 Consuming results

FAA's output is standard SARIF 2.1.0, deliberately kept usable across multiple consumers. Every other format FAA writes (the SSC upload zip, the Markdown reports) is derived from that SARIF, either during the scan or afterwards with `-convert`; Section 4.5 explains the model and Section 4.5.5 the Markdown formats. This chapter describes how to bring the results into each platform.

8.1 How Fortify Agentic Analyzer structures its SARIF

SARIF draws a sharp line between a *rule* (the kind of issue a tool looks for) and a *result* (a place where the tool found one). That line fits a traditional SAST engine exactly: it ships a fixed rule set, each rule carries a description and remediation advice written once by the vendor, and every result points back at the rule it matched.

FAA does not work that way, in two respects. It has no fixed rule set: what it looks for is driven by prompting an LLM over your code, not by a catalog of patterns. And, more importantly, the explanation and the remediation guidance are written for that finding: naming the function it found, the parameter that is unsafe, the framework you actually use, and the fix that makes sense in that code. That specificity is a large part of what FAA is for. Canned per-category text is what traditional SAST already gives you; FAA is worth its cost precisely because its guidance is not canned.

Those two facts collide with a limitation of the format. A SARIF result has exactly one field for human-readable prose: its `message`, intended for a one-line statement about the finding. The fields that hold a full explanation and remediation guidance, `fullDescription` and `help`, exist only on the rule. So there is no way to attach a finding-specific explanation and a finding-specific fix to a SARIF result directly: whatever channel you choose, those fields live on the rule.

FAA therefore emits **one rule per finding**, strictly one-to-one. Each rule carries that finding's own explanation, its own remediation guidance, its category and Fortify kingdom, and its severity scores. The rule's identifier is the category followed by the finding's stable instance id, for example `sql-injection/0CB3F319CFB77AFD08B8E6E871BD77EB`.

8.1.1 What this means in practice

For the consumers FAA targets, this is what makes the output work rather than something to work around:

- *The detail arrives where each platform already looks for it.* FoD reads its Explanation from `fullDescription` and its Recommendations from `help`; SSC's parser plugin reads the same two fields. Because those fields are per finding, so is what you read on screen, with no platform-side configuration.
- *Categories are still first-class.* The Fortify category and subcategory travel in their own fields on every rule, so grouping, filtering, and reporting by category work normally in FoD and SSC. The one-to-one rule identifier does not replace the category; it sits alongside it.
- *Findings are still trackable across scans.* The rule identifier ends in the finding's stable instance id, which is derived from the location and category rather than from the run. Re-scan unchanged code and the same finding keeps the same identifier. "One rule per finding" does not mean "a new rule every scan".
- *The output remains standard SARIF 2.1.0.* Nothing in the specification requires a rule to be shared between results or to have existed before the scan; a rule identifier is defined simply as a stable, opaque identifier. FAA's output validates against the official SARIF 2.1.0 schema.

8.1.2 The trade-off

There is a real cost, and it is worth stating plainly. Rule identity is the axis that generic tooling tends to use for operations on a class of issues: grouping an alert list by rule, configuring "ignore this rule", or setting per-rule thresholds. When every finding is its own rule, those views degenerate to one entry per finding. A third-party SARIF consumer that groups by rule identifier will show groups of

one.

FAA accepts that trade-off deliberately. Its output targets Fortify on Demand and Fortify SSC; neither keys on the rule identifier for category grouping, and both display the finding-specific explanation and remediation correctly under this model. The alternatives all cost more: folding everything into the one-line message destroys the structure the platforms render into separate fields, and inventing custom properties to hold the detail means no consumer displays it at all until each one is taught to.

8.1.3 Which code was scanned

When the scanned directory is inside a git repository, FAA records the repository state in the standard SARIF `versionControlProvenance` field: the commit (`revisionId`), the branch (left out on a detached HEAD, as is common in CI), a tag pointing at the commit (`revisionTag`), and the remote URL (`repositoryUri`). The field's properties add all tags at the commit, the git describe label, the commit time, the scanned directory's path within the repository, and whether the working tree had uncommitted changes (`dirty`, with counts). Credentials embedded in a remote URL are never written.

Git is optional for this: without git on the PATH, or outside a repository, the field is simply omitted. The state is captured when the scan starts. If the scan is interrupted and HEAD moves before it is resumed, FAA warns, keeps the original commit, and sets `revisionChangedDuringScan`, because the work done before the interruption analyzed the earlier commit.

8.2 Fortify on Demand (FoD)

FoD supports FAA results natively via SARIF import; no plugin is required. Import the SARIF file (by default `faa.sarif`) into the relevant release through FoD's SARIF import. Because FAA emits complete, standard SARIF fields, the findings appear with their rule metadata, severity, and locations intact.

8.2.1 Full detail in the FoD issue view

FoD renders the finding's plain-text channels in its issue view: the one-line summary (`message.text`) and the description on the *Vulnerability* tab, the remediation under *Recommendations*, and the code snippet on the *Code* tab. Because FoD renders those plain-text channels as HTML (and does not surface the Markdown variants), FAA's default message format (`--message-format fod`, Section 4.5.7) shapes them as HTML so they display well. No configuration is needed: it is on by default.

If the same SARIF also goes to a consumer that displays those plain-text channels verbatim instead of rendering them, run with `--message-format plain` for that copy, so no markup shows up raw. This choice affects only the plain-text channels (`message.text` and the rule texts); SSC is unaffected either way, since the plugin renders the Markdown variants.

Remember that, as described in Section 3.1, the FAA scan runs locally even when you use FoD; only the resulting SARIF is brought into FoD.

8.3 Fortify Software Security Center (SSC)

SSC consumes FAA results through a dedicated parser plugin, which renders them under the engine type `FORTIFY_AA`. You must install this plugin in SSC before uploading FAA results. The plugin is available at [fortify/fortify-ssc-parser-faa](https://github.com/fortify/fortify-ssc-parser-faa).

Once the plugin is installed, run the scan with `--output-format sarif,zip` so it produces the upload artifact (by default `faa.zip`) alongside the SARIF, and upload that zip to the appropriate application version, exactly as you would upload other scan results. The plugin reads the SARIF and presents FAA findings with their Fortify categories, severities, and remediation detail.

If you have already scanned and did not ask for the zip, you do not need to scan again. Produce it from the SARIF you have:

```
fortifyaa -convert faa.sarif --output-format zip
```

9 Agentic safety

Running an autonomous agent over your source code raises a fair question: how do you know the agent (or the LLM behind it) will not modify your code or upload it somewhere? This chapter explains FAA's safety design, how it is enforced for each agent backend, and where its limits are, so you can decide what additional isolation, if any, your environment requires.

9.1 The safety principle: read-only and offline

FAA drives the agent over a source tree it does not own and should not trust (your code). The governing principle is therefore simple and non-negotiable:

- *The agent must not modify anything.* The code under analysis stays exactly as you provided it.
- *The agent must not reach the network.* No fetching URLs, no web search, no exfiltration channel of its own.

The only outbound traffic during a scan is the LLM provider call itself (your Bedrock/Anthropic/claude endpoint) and the Fortify license check, both made by FAA's deterministic code, never by a tool the agent can invoke.

9.2 How it is enforced: an allow list of tools

An agent can only do what its registered *tools* let it do. FAA registers only read-oriented tools (read a file, search, list) plus its own schema-validated structured tools (report a finding, validate a finding). No write tool, no shell tool, and no network or fetch tool is ever registered. FAA deliberately uses an allow list rather than a deny list: a newly introduced built-in (a future fetch variant, a new editor tool) is not granted unless it is added on purpose.

9.2.1 Pi (bundled, default): an in-process capability boundary

The Pi agent runs in-process inside FAA. Its session is created with an explicit, closed list of tools: only what FAA passes exists in the session, so there is no ambient catalog of tools for the agent to escape to. The read-only built-ins are limited to reading, searching, finding, and listing files; `edit`, `write`, and `bash` are never included, and Pi exposes no network tool that FAA registers. FAA further supplies its own system prompt and suppresses any agent-instruction files that might be present in the scanned tree, so the code being analyzed cannot rewrite the agent's instructions.

9.2.2 Claude Code (external, opt-in): deny-by-default controls

Claude Code is a separately installed CLI that FAA runs as a child process. For each stage FAA spawns the `claude` binary in headless mode (`claude -p`, with the prompt on standard input and `--output-format stream-json` for progress), passing a fixed set of command-line flags that lock the run down. The security-relevant flags are:

- *Deny-by-default permissions* (`--permission-mode dontAsk`): anything not explicitly allowed is denied, without prompting.
- *An explicit allow list* (`--allowedTools`) of the read tools (`Read`, `Grep`, `Glob`) plus this run's structured tools (`mcp__faa__report_finding` and the like). `Web fetch` and `web search` are not on it, so they are denied.
- *An explicit denial* (`--disallowedTools`) of the mutating tools (`Write`, `Edit`, `Not ebookEdit`, `Bash`) as a redundant second layer.
- *Isolation of MCP configuration* (`--mcp-config` together with `--strict-mcp-config`): the user's other configured MCP servers are ignored entirely, so a network-capable MCP server in a global config cannot leak into the scan. FAA points Claude Code only at its own in-process MCP server, bound to localhost on a random port with a per-run bearer token, which serves only the structured reporting tools.
- *Scoped reads* (`--add-dir`): reads are limited to the project tree, not the whole filesystem. In addition, `--no-session-persistence` ensures that no transcript of the analyzed code is written to disk.

These properties are locked down by regression tests that assert against the

same tool lists the product uses, so the guarantee cannot silently drift as the code evolves.

9.3 Scope and limits of the guarantee

It is important to be precise about what this design does and does not guarantee.

What it guarantees. The agent's reasoning loop has no tool with which to modify your source tree or make its own network requests. For Pi (in-process) this is a genuine capability boundary. For Claude Code it is enforced by the CLI's permission system. Either way, even if text injected from the scanned code tried to convince the agent to "write a file" or "fetch a URL," there is no tool through which it could act. The allow list bounds the blast radius of any prompt-injection attempt.

What it does not (yet) guarantee. This is enforced at the agent's tool layer, not by an operating-system or container sandbox around the host process. In particular, the external `claude` subprocess runs with the ambient filesystem and network access of the user who launched the scan; FAA relies on the CLI honoring its deny-by-default settings rather than on a kernel-level jail. This is a deliberate, documented limit.

9.4 Do you need to run Fortify Agentic Analyzer in a sandbox?

Our guidance:

With the default Pi agent — generally, no. Pi runs in-process with a closed tool list, which is a real capability boundary: the agent has no tool to write your code or send it anywhere. For most environments this is sufficient, and a separate sandbox is not required.

With the Claude Code agent — consider it, depending on your assurance requirements. Because Claude Code is an external process with the launching user's ambient access, the network/filesystem guarantee rests on the

CLI's permission controls rather than on OS-level enforcement. If your organization requires a hard guarantee that network egress is impossible (as opposed to merely not offered to the agent as a tool), run the scan inside an OS- or container-level egress restriction (for example, a network-isolated container with only your LLM endpoint and Fortify endpoint reachable).

In short, the higher your required level of assurance, the more it makes sense to add an external sandbox on top of FAA's built-in controls, especially when using the external Claude Code backend.

10 Performance and results

This chapter sets expectations for what an FAA scan costs, how long it takes, how repeatable its results are, how complete its coverage is, and what it has actually detected on real-world vulnerable code. Ideally a scan would be fast, cheap, and perfectly deterministic. In practice there is a trade-off between those goals, and FAA gives you policy controls to position yourself along it.

About the figures in this chapter. What a scan costs, how long it takes, and how much it finds are not fixed properties of FAA. They move as FAA evolves and as the underlying models do (a new model generation can shift both cost and detection substantially), and they depend heavily on how you configure the scan, since the policy file admits a wide range of legitimate configurations with very different economics. The numbers below are measurements of particular projects under particular settings, and they represent our best characterization at the time of writing rather than committed benchmarks. Read them for the shape of the trade-offs, and measure your own codebase under your own policy before drawing conclusions about it.

10.1 Cost and speed

FAA scans make real, paid LLM calls, so cost is a first-class concern. To make it concrete, this section reports two scans of open-source applications, run in September 2026 with FAA 26.4 and its shipped default policy, against Claude models on AWS Bedrock.

Treat them as examples, not as a prediction for your own code. A scan's cost and duration depend on the code itself, on the policy, on the models and what they cost at the time, and on your provider's throughput. What makes these particular numbers worth reporting is the first project: Apache Tomcat is a large, real-world codebase of the size of a substantial business application, not a small demonstration project, so its figures show what a full scan of serious software

looks like. The second project, Healthchecks, the cron-job monitoring service behind healthchecks.io, is about a tenth of the size and in a different technology, for contrast. The following table summarizes both scans.

	Apache Tomcat 11.0.26	Healthchecks 4.4
Application	Java web server	Python/Django web application
Size	530 kLOC in 4,430 files	68 kLOC in 992 files
Files analyzed	2,514	446
Elapsed time	31 min 12 s	20 min 37 s
Estimated LLM cost	\$92.83	\$15.39
Cost per analyzed file	\$0.037	\$0.035

The cost splits over the three pipeline stages as follows (Claude models; tokens in millions):

Stage	Model	Input	Output	Cache read	Cache write	Cost
<i>Tomcat</i>						
Plan	Haiku 4.5	0.38	0.37	—	—	\$2.24
Analysis	Sonnet 5	<0.01	1.11	110.15	8.29	\$80.74
Validation	Opus 5	<0.01	0.04	0.93	0.28	\$9.85
<i>Healthchecks</i>						
Plan	Haiku 4.5	0.08	0.08	—	—	\$0.47
Analysis	Sonnet 5	<0.01	0.29	14.00	0.89	\$11.90
Validation	Opus 5	<0.01	0.01	0.66	0.06	\$3.01

Sizes are lines of code across all languages in the repository, excluding blank lines and comments. Not all of those files are analyzed: the policy's file filters leave out images, archives, lock files, build output, git-ignored files, and the like, which is why the number of files analyzed is lower. Costs are FAA's own estimate, computed from the token usage the provider reports and the models' list prices;

your actual bill may differ, for example with negotiated pricing.

A few things stand out:

- *Cost follows the number of files analyzed.* Tomcat has eight times the lines of code and five to six times the analyzed files of Healthchecks, yet both came out at about 3.5 to 4 cents per analyzed file. Two projects do not make a rule, but the number of files that pass the policy's filters is a better first estimate of cost than lines of code.
- *Analysis dominates.* It accounts for 87% of the Tomcat cost and 77% of the Healthchecks cost. The plan stage, on a small model, is cheap; validation's share depends mostly on how many findings there are to review.
- *Almost all input is served from the prompt cache.* As the agent works through a batch, each turn resends the conversation so far, and the provider serves the repeated part from its cache at a fraction of the normal input price. This is what keeps a deep, many-turn analysis affordable, so a provider and model with prompt caching matter for cost.
- *Elapsed time grows much more slowly than size.* Tomcat took about 1.5 times as long as Healthchecks for 5.6 times as many analyzed files, because batches are analyzed concurrently. Added up, the models worked for about 5.8 hours on Tomcat and 1.6 hours on Healthchecks. How far concurrency compresses that depends on your provider's rate limits.
- *The default budget cap is \$50.* Healthchecks stayed within it. For Tomcat, the cap was raised on the command line with `-Dmaxcost=200`, in the knowledge that the scan would exceed \$50. Without that, the scan would have stopped at the cap, and could then have been resumed with a higher one (Section 4.6).

If you run FAA through a Claude Code subscription and have not enabled spending beyond your plan's usage limits, a scan has no marginal cost: the subscription has already been paid.

You have several levers to manage cost, most of them in the policy file (Chapter 6):

- `maxcost` — a USD cap that stops the scan from overrunning your budget. The first analysis batch runs alone to calibrate the cost per batch; further batches are dispatched only while the projected total (including work in flight) stays within the cap. A capped scan is resumable, so you can raise

the cap and continue where it stopped. Enforcing the cap requires knowing what each turn costs: if a model's cost cannot be determined (an unlisted model whose provider reports no usable token/cost data), a capped scan stops rather than spend unaccountably. Scan such models with `maxcost=0` (explicitly uncapped), or give the model cost metadata (for example, in its `pi_models.json` entry).

- Cheaper models per stage, for example a cheaper `analysis.model.family` or `validation.model.family`.
- Toggling stages, for example skipping validation, at the cost of output quality.
- Scoping (`--scope`) to the most security-relevant code.
- Change-driven scans (`--incremental` / `--differential`) so you only pay to analyze what changed (Section 4.8).

Token usage and estimated cost are printed per turn and summarized per stage at the end of every run, and written into the SARIF metadata.

10.2 Determinism and repeatability

AI is inherently non-deterministic. An LLM does not guarantee the same output twice for the same input, and FAA cannot change that. What FAA does do is push determinism as far as it reasonably can:

- The harness, not the agent, owns everything reproducible: file enumeration, batching, deduplication, category alignment, fingerprints, snippets, and SARIF formatting.
- Sampling settings are pinned wherever the agent allows it.
- The `analysis.samples` property lets you run multiple analysis passes and union the results, reducing run-to-run recall jitter on borderline findings (Chapter 6).

The practical outcome we observe is that repeatability can be made strong on the real, important security findings, while some non-determinism is expected to remain on “debatable” findings near the threshold of what counts as an issue. We treat determinism as a best-effort engineering goal, something we actively work to maximize, and not as a guarantee.

10.3 Completeness

Completeness of *coverage* is a property of FAA's driver, not of the agent's memory: FAA deterministically enumerates every in-scope file and ensures each is reviewed, rather than trusting the agent to remember to look everywhere. This is why the harness, and not the agent, decides what gets analyzed.

Completeness of *detection* (how many of the real issues in the reviewed code are actually found) depends heavily on the model used. More capable models detect more. Because FAA is bring-your-own-LLM and model selection is configurable, you can plug in more powerful models as they become available, and detection improves accordingly. The next section reports what detection looks like in practice on real vulnerable code.

10.4 Detection results on real CVEs

Cost and coverage figures say nothing about whether the analysis finds real vulnerabilities. To measure that on the flaw classes FAA targets, we assembled a test set of twelve recent, real CVEs and ran FAA against the affected projects. The design of the test set matters as much as the score, so we state it explicitly:

- All twelve are real vulnerabilities in real, production open-source projects: large, messy, real-world codebases, not toy examples or synthetic benchmark suites.
- All were published in February 2026 or later, after the training cutoff of the underlying model. A detection therefore cannot be the model remembering the CVE from its training data; it has to come from analyzing the code.
- Each case is pinned to the exact vulnerable commit, and every hit was verified by hand against both the vulnerable and the fixed source. A "hit" means FAA reported the actual flaw at its actual location, not something vaguely nearby.
- The cases deliberately span ten languages and ecosystems.

These runs used `analysis.samples=2` rather than the default of 1: each project was analyzed in two independent passes and the findings of both were combined, which makes detection more consistent from run to run at about twice the cost of the analysis stage (Chapter 6). A scan at the default setting may miss

a finding that one of the two passes caught.

In this exercise, with the core analysis agent running on Claude Sonnet 5, FAA detected ten of the twelve:

CVE	Project (language)	Vulnerability	CVSS	Found
CVE-2026-29000	pac4j (Java)	JWT signature verification bypass	10.0	yes
CVE-2026-54782	CoreWCF (C#)	SAML signature validator that never validates	10.0	yes
CVE-2026-49448	authentik (Python)	Broken enrollment logic bypasses authentication	9.8	yes
CVE-2026-28514	Rocket.Chat (TypeScript)	Password-check result never awaited	9.8	yes
CVE-2026-48611	phpBB (PHP)	Attacker selects which authentication provider runs	9.8	no
CVE-2026-55518	Avo (Ruby)	Missing authorization on nested create	9.6	yes
CVE-2026-44351	fast-jwt (JavaScript)	Empty HMAC secret accepted as valid	9.1	yes
CVE-2026-5194	wolfSSL (C)	Certificate signature-algorithm / key-type confusion	9.1	no
CVE-2026-41076	Request Tracker (Perl)	LDAP unauthenticated bind: empty password logs you in	8.1	yes
CVE-2026-11404	Mongoose (C)	Out-of-bounds read parsing a TLS ClientHello	7.5	yes

Continued on next page

CVE	Project (language)	Vulnerability	CVSS	Found
CVE-2026-28815	swift-crypto (Swift)	Missing length check leading to an out-of-bounds read	7.5	yes
CVE-2026-54318	Home Assistant (Kotlin)	Exported receiver trusts forged intents	7.1	yes

Most of these are exactly the kind of flaw traditional data-flow SAST is not built to see: a required check that is simply absent, a validator that computes a result and then ignores it, an authorization decision that trusts the wrong source, an asynchronous result that is never awaited. That is the complementary role FAA is designed for (Chapter 1).

The two misses are instructive about limits. In phpBB, the vulnerable code path is never referenced statically: the authentication provider is chosen dynamically at runtime from attacker-supplied input, so no reading of the code's static call structure surfaces it. In wolfSSL, the flaw sits in a single, very large cryptographic source file, and the analysis spread its effort broadly across that file rather than converging on the one defective check.

Read these numbers with the appropriate caveats. This is a small test set (twelve cases) that we assembled ourselves; it is not an independent benchmark. The cases were selected to represent the high-severity logic- and design-flaw classes FAA targets, so the detection rate should be read as "FAA finds this class of bug in real code", not as a general recall figure across all vulnerability classes, and it says nothing about false-positive rates. FAA was also actively improved during the exercise (working through real code informed tuning of the analyzer itself), so the figures describe the current tool on these cases, not a frozen tool on a held-out set. And as everywhere in this guide: detection quality depends on the model; a different model choice will give different results.

Separately from the twelve known CVEs, working through these projects also surfaced several previously unknown vulnerabilities. These were reported to the affected projects through responsible-disclosure channels and are not further documented here. A more detailed write-up of the whole exercise is published

on the OpenText Cybersecurity blog.¹

¹*Applying Agentic Analysis to Real AppSec Problems*, <https://community.opentext.com/cybersec/b/cybersecurity-blog/posts/applying-agentic-analysis-to-real-appsec-problems>.

11 Troubleshooting

This chapter describes where FAA writes its logs and what to include when you report a problem to OpenText.

11.1 Locate the log files

FAA keeps its files in the Fortify working directory: %LOCALAPPDATA%\Fortify\faa26.4 on Windows, and ~/.fortify/faa26.4 elsewhere. The following files are useful when you investigate a problem:

Run log — log/faa.log in the working directory. By default it holds the timestamped halt and error entries of every run. With --log <path>, a scan writes its run log to <path> instead, and mirrors into it everything it prints to the console, so the file is a complete record of that run.

Debug log — debug.log in the scan's build directory, written only when you run the scan with --debug. It is a timestamped timeline of what every batch did: dispatch, the agent's tool calls, each LLM request and response, completion, cost, and budget decisions.

Agent transcripts — the transcripts directory next to debug.log, also written with --debug. It holds the full conversation of every agent run.

LLM failure dumps — the llm-failures directory next to debug.log. When an LLM call fails in a --debug run, FAA writes the exact request it sent and the HTTP response it received to a file here.

Each scan has its own build directory under build in the working directory; Section 5.8 explains how it is named. A build directory stays in place after the scan completes, until you remove it with -clean.

11.2 Report issues and request enhancements

To report an issue or request an enhancement, contact OpenText Support at <https://www.microfocus.com/support>. Include the following information:

- Product: OpenText Fortify Agentic Analyzer
- Version: the first line that any `fortifyaa` command prints, for example `fortifyaa -help`
- Platform and operating system (for example, Windows 11 or Ubuntu 24.04)
- The coding agent, LLM provider, and models that a scan uses: the output of `fortifyaa -selftest --no-live`, which makes no paid LLM calls
- The policy file you used, and any `-D` overrides (the effective configuration is also recorded in the SARIF output)

To request an enhancement, describe the feature and what you want to use it for.

To report an issue, give the command line you ran and enough detail for OpenText Support to reproduce the problem. The more precisely you describe it, the faster Support can analyze and resolve it. Also include the relevant log files, or the relevant parts of them, from when the problem occurred. If a scan hangs or an LLM call fails, run it again with `--debug`, and include `debug.log` and any files in the `llm-failures` directory.

Review diagnostic files before you send them. The agent transcripts and the LLM failure dumps contain the prompts sent to the model, including excerpts of the source code being scanned. Share only what your organization's policies allow you to share.



© Copyright 2026 Open Text