AETHER

Technology Monitoring & Hard Problem

High-performance memory-safe web browser

Contents

Introduction 2

Part 1: The Hard Problem 2

- 1.1 Problem statement 2
- 1.2 What makes it hard 2
- 1.3 Naive approach failure 2
- 1.4 Quantified success criteria 2

Part 2: Technology monitoring setup 3

- 2.1 Sources followed 3
- 2.2 Tools and automation 3
- 2.3 Cadence 3
- 2.4 Split across the team 3
- 2.5 Monitoring log 4
- 2.6 Internal sharing 4

Part 3: State of the art on the hard problem 4

Part 4: Experimentation — 2 prototypes 5

- 4.1 Decision prototype (due end of January 2027) 5
- 4.2 Exploration prototype (between February and April 2027) 5

Part 5: Decision & integration 5

Part 6: Security tracking and actions 5

Part 7: Participation in technical communities 6

Introduction

This document covers mandatory objective #1 of the Technical Track: organizing the team's technology monitoring and using it to solve Aether's hard problem. It is written to be reviewed with the mentor (Jean-Philippe Roze) and then completed progressively across the mentoring sessions (every 3 weeks), following the EIP guide's calendar.

Status: initial draft, covering the deliverables due end of October 2026 (hard problem characterization + monitoring setup). The following sections (state of the art, prototypes, decision, integration, communities) are outlined and will be filled in at the deadlines indicated below.

Part 1: The Hard Problem

1.1 Problem statement

Aether natively embeds an ad and tracker blocking engine (the **components/net/adblock** module), already highlighted as a core product differentiator (Budget Analysis, Context Audit, Action Plan §2.1). Today this engine is not yet built: the team does not yet know how to filter a network request against an EasyList-format rule set (tens of thousands of rules) without adding perceptible latency to page loading.

1.2 What makes it hard

- **Volume** - EasyList/EasyPrivacy lists contain tens of thousands of rules, and a modern web page typically fires 50 to 150 requests to filter.
- **Time** - filtering must stay invisible to the user: it runs on the critical path of page loading (see also Objective 2, critical path).
- **Compatibility** - the EasyList format mixes domain rules, cosmetic filters and exceptions; a partial implementation either breaks sites (false positives) or lets trackers through (false negatives).
- **Reliability** - rule lists keep evolving; the engine has to stay correct as they grow.

1.3 Naive approach failure

A naive implementation (sequentially comparing every request against every rule, plain substring matching) has a cost that grows linearly with the number of rules per filtered request. With tens of thousands of rules and dozens of requests per page, the cumulative cost quickly becomes the bottleneck of page loading, directly undermining Aether's speed promise.

1.4 Quantified success criteria

Proposed targets, consistent with the ones already set in the Action Plan (milestone M2), to validate with the mentor:

- Blocking rate ≥ 95% on the EasyList corpus (10,000+ rules), matching the M2 milestone commitment.

- Average processing time per request ≤ 0.5 ms on that same corpus.
- No measurable functional regression on a reference sample of sites (no false positive breaking primary content).

Part 2: Technology monitoring setup

2.1 Sources followed

Source	Type	Why we follow it
This Week in Rust	Weekly newsletter	Weekly overview of the Rust ecosystem
Official Servo blog (servo.org/blog)	Blog	Progress on the rendering engine Aether directly builds on
GitHub releases - Servo, egui, tokio	Repository tracking	Alerts on breaking changes and new features in key dependencies
Reddit r/rust, r/programming	Forum	Community feedback, experience reports on similar problems
internals.rust-lang.org	Specialized forum	In-depth discussions on the evolution of the language and ecosystem

List carried over from the Action Plan (§3.1), the team's mandatory monitoring section. To extend with sources specific to the hard problem as the state of the art develops (e.g. the adblock-rust repository, content-filtering forums).

2.2 Tools and automation

- GitHub watch enabled on the Servo, egui, tokio and adblock-rust repositories, notifications centralized in a dedicated channel.
- RSS aggregator (e.g. Feedly) on This Week in Rust + the Servo blog.
- Dedicated Discord/Slack channel for monitoring, fed by every member.
- GitHub wiki page for the monitoring log (see 2.5).

2.3 Cadence

Proposal: roughly 30 minutes per week and per member, spent on their assigned monitoring area (see 2.4), with a collective debrief at least once a month (see 2.6).

2.4 Split across the team

Proposed split by area, to validate and assign as a team:

Monitoring area	Assigned member
Rust ecosystem & performance (cargo, clippy, tooling)	Axel
Servo (rendering engine internals, layout, WebRender)	Axel
egui / user interface	Axel, Mattéo
Networking, security, filtering (adblock, TLS)	Axel
MPV / video & technical communities	Corentin, Nathan
Devtools	Thomas

2.5 Monitoring log

Format proposed: one line per entry, fed continuously from October to May (20 entries minimum). Format examples below (to be replaced with real entries):

Date	Source	Topic	Potential impact / change	Action taken or reason for inaction
[example] 03/10/2026	GitHub - adblock-rust	Rust filtering library open-sourced by Brave	Direct candidate for the decision prototype	Added to the state of the art (see 3.1)
[example] 10/10/2026	Servo blog	Progress on WebRender GPU rendering	Could inform the exploration prototype	To keep watching, no action yet

2.6 Internal sharing

Proposed ritual: a 15-minute slot at the start of a team meeting, at least once a month from November 2026 (6 sessions minimum over the year). Each session is logged (who presented what, to whom, what was taken from it) on the wiki.

Related requirement: track at least 3 pieces of monitoring integrated into the project (a library adopted, a practice reused...), with the original source and what it changed.

Part 3: State of the art on the hard problem

First pass, to be deepened with dated sources before end of December 2026 (see calendar).

Approach	Performance	Maturity	Integration	License
Custom engine (Rust, purpose-built matching)	To measure	To build from scratch	Full control, but significant dev effort	N/A
adblock-rust (Brave, open source)	Reputed very fast (used in production at Brave)	Mature, actively maintained	Rust crate, direct integration possible	MPL-2.0

Approach	Performance	Maturity	Integration	License
uBlock Origin-style approach (algorithmic reference, JS)	Very high performing in its own ecosystem	Very mature	Not directly reusable (JS), serves as a design reference	GPL-3.0

Part 4: Experimentation — 2 prototypes

4.1 Decision prototype (due end of January 2027)

- Compare adblock-rust (leading candidate) against a simple custom/naive baseline, on the same rule corpus and the same request set.
- Same protocol for both: same environment, same measurements (blocking rate, processing time per request).
- Code for both prototypes kept and accessible in the repo.

4.2 Exploration prototype (between February and April 2027)

- Candidate idea: GStreamer vs. MPV for video playback (retroactively validate the MPV choice already made in the Action Plan), or a Servo subsystem the team doesn't yet master (layout parallelism, WebRender).
- Hypothesis to state before starting; the result counts even if negative.

Part 5: Decision & integration

- Documented technical decision (end of February 2027): approach chosen, approaches set aside and why, consequences accepted.
- Real integration (end of April 2027): identifiable commit/merge, before/after measurement against the criteria in section 1.4, acknowledged limits.

Part 6: Security tracking and actions

- cargo-deny checks the Rust dependency graph against the RustSec advisory database (deny.toml, section [advisories]). That database covers known vulnerabilities (CVE / RUSTSEC identifiers). Advisories listed under ignore are accepted on purpose (unmaintained crates we cannot replace yet).
- Every pull request runs this check. The default CI profile for a pull request is linux-unit-tests lint (.github/workflows/main.yml). Lint runs ./mach test-tidy, and tidy runs cargo-deny --all-features check (python/tidy/tidy.py).
- A vulnerability that is not ignored is an error. Tidy fails, the Lint job fails, and the aggregate check Result exits 1. With Result required to merge, the pull request stays blocked until the

advisory is fixed or explicitly ignored.

Part 7: Participation in technical communities

From November 2026 to May 2027 — at least 2 active communities, 5 public contributions, 1 accepted external contribution.

- Proposed target communities: r/rust, a Rust Discord server (French-speaking or international), the Servo forum/tracker.

Document Date: September 16, 2026

Modify Date: November 13, 2026