SignalMTA User Guide

Release 0.1.2-rc4 · October 3, 2026

Self-hosted email delivery infrastructure. Configuration-first WebUI, real-time SMTP logging, CLI/API metrics, signed webhooks, and BigStats.email reporting.

Product map

Before you touch a single config key, get a feel for how SignalMTA is put together.

SignalMTA is organized around the questions that actually come up in production: who's submitting mail, which tenant or stream owns it, which IP it should leave from, how the destination is responding, what the queue is doing right now, and what you as an operator should do about it.

- An MTA you run yourself
- A durable queue
- Intelligence on the destination side
- Control over IPs and routes
- A WebUI, API, and CLI
- Help from SignalAI
- The short version, if you're in a hurry

SignalMTA takes mail in over SMTP or API, checks who's sending it, tags it with tenant and stream context, writes accepted recipients to durable storage, runs routing and traffic-shaping policy against it, tries delivery, logs the exact SMTP and DSN evidence, and gives you control through the WebUI, API, CLI, and Signal Scripting Language.

How to read this guide

Orient yourself here first, then work through the getting-started path. The deeper sections cover the same feature areas you'll actually use day to day: submission, authentication, queues, traffic shaping, IP management, bounces, exports, clustering, and performance tuning.

Submission Control

Take mail in through authenticated SMTP submission, trusted relay paths, or API injection, and hang onto the metadata you'll need later for routing, logging, suppression, and compliance.

- SMTP submission plus controlled relay intake
- Handling for tenant, stream, tag, and custom headers
- List-Unsubscribe and SignalMTA's own control headers
- Keeping the queue intact

Once mail is accepted, keep it recoverable and prioritized, so a bulk backlog, a throttling destination, or a retry-heavy domain can't starve your high-value streams.

- Ready, scheduled, held, deferred, and final states for every message
- Queue admission and retry lanes that respect priority

- A drilldown path from queue pressure straight to message evidence
- Knowing what the destination is doing

See what recipient domains and MX groups are doing right now, complete with exact DSN strings, acceptance trends, which streams are affected, and suggested next steps.

- Views by recipient domain and remote MX
- Context on customer stream, tenant, pool, and source IP
- Verbatim SMTP and DSN evidence you can actually troubleshoot with
- Shaping traffic

Control delivery at the level of domain, MX, tenant, stream, IP pool, source IP, or error class, using adaptive policies that only slow down the scope that's actually causing trouble.

- Caps, floors, bursts, concurrency limits, and retry schedules
- Adjustments that kick in automatically on 4xx and 5xx patterns
- Manual overrides for when you need a scoped exception
- Authentication and trust

Walk operators through SPF, DKIM, double DKIM signing, return-path alignment, TLS certificates, and readiness checks before you ramp up volume.

- Domain setup and DNS record generation right from the WebUI
- Controls for selector rotation and signing behavior
- TLS policy, how STARTTLS behaves, and certificate automation
- Reporting and exports

Give operators live visibility inside SignalMTA itself, while still supporting external observability and deliverability platforms for teams that want them.

- Records of delivery attempts, SMTP sessions, bounces, FBLs, and webhooks
- Reports you can window by time, across domains, streams, tenants, and IP pools
- Prometheus-compatible metrics plus structured event export
- Different ways to run SignalMTA
- The WebUI

Reach for this when you're doing live operations, troubleshooting, guided setup, reviewing traffic shaping, running sender-authentication workflows, checking SignalAI advisories, or looking at role-based views for CTO, engineering, and deliverability teams.

API

Reach for this when you're provisioning tenants, streams, domains, IP pools, webhooks, and policy updates from an internal platform or a service provider's own control system.

CLI

Reach for this for install automation, configuration validation, health checks, release verification, batch operations, support bundles, and recovery workflows.

Signal Scripting Language

Reach for this when your team needs precise policy logic at the submission, routing, authentication, retry, bounce, logging, or delivery decision points.

- A suggested order to read things in
- Planning your evaluation

Work through the evaluation guide, sizing guidance, and install configurator to pick a realistic first host and test path.

Secure intake

Lock down where relay traffic can come from, SMTP submission, API authentication, TLS, user accounts, OTP, and audit requirements.

Authenticate domains

Create your domains, publish DNS, verify DKIM/SPF/DMARC/return-path alignment, and get list-unsubscribe behavior ready.

Shape delivery

Set up IP pools, routes, throttles, retry rules, warm-up schedules, and destination-specific controls.

Operate and prove

Lean on live queues, destinations, bounces, SMTP logs, SignalReports, exports, and support bundles to confirm you're ready for production.

Getting Started

In 0.1.2-rc4 the included WebUI is dedicated to configuration management and real-time SMTP operational visibility. Historical reporting, trend analysis, bounce and deferral analysis, source-IP performance, provider/domain reporting, and long-window analytics are handled by BigStats.email, which is purchased separately from the SignalMTA license.

SignalMTA remains fully observable without BigStats: current metrics are available through ./bin/signalmta metrics and the runtime APIs, and delivery telemetry is exportable through signed HTTP POST webhooks.

SignalMTA is email delivery infrastructure you download and run yourself. Install it on Linux, apply the license key SignalMTA gives you, set up your domains and routes, then run delivery day to day through the included WebUI, CLI, API, and Signal Scripting Language.

The screenshots throughout this guide come straight from a live SignalMTA WebUI. Wherever you see a code block or table, it's documenting the same settings you'd find in the console, CLI, and API.

If SignalMTA is new to you, work through this guide in order: figure out your installation goals, learn the core terms, pick an evaluation path, size the host, install and license the software, get comfortable with the WebUI and configuration model, then move into the feature areas — submission, domains, queueing, traffic shaping, IP strategy, bounces, logging, security, clustering, and performance.

Start on a controlled evaluation host before you send anything that matters. Confirm queue durability, DNS performance, TLS negotiation, domain authentication, relay controls, bounce handling, logging exports, and recovery behavior first. Treat that first week as qualifying your infrastructure: you're validating the host, network, DNS, IP plan, delivery policy, and how your team actually operates it.

SignalMTA won't march every evaluator through a beginner wizard. Expert mode is the default, so if you've run an MTA before you can go straight into the WebUI, API, CLI, config files, and Signal Scripting Language. Guided setup and import workflows are there if they help, but the usual readiness checks stay advisory unless a setting would create a real safety, license, queue-integrity, or live-delivery blocker.

Provision a supported Linux server with NVMe-backed spool storage and outbound network access you can count on.

Install SignalMTA, then bootstrap your first administrator account with OTP-based console access.

Apply the SignalMTA license key through the WebUI or CLI and confirm the node comes up active.

Create a sending domain, publish DKIM/SPF/DMARC/return-path records, and check DNS from more than one resolver.

Set relay authorization so only sources you've approved can submit mail over SMTP or API.

Build out IP pools, routes, stream policies, queue priorities, rate limits, retry rules, and webhook exports.

Run the readiness and performance qualification commands before you point any public internet traffic at it.

```
./bin/signalmta bootstrap-admin --config /etc/signalmta/signalmta.json
./bin/signalmta license apply --key "$SIGNALMTA_LICENSE_KEY"
./bin/signalmta evaluation status --config /etc/signalmta/signalmta.json
./bin/signalmta domain-auth-plan --domain example.com --selector sig1 --bounce-domain bounces.example.com
./bin/signalmta config validate --config /etc/signalmta/signalmta.json
./bin/signalmta completion-audit --config /etc/signalmta/signalmta.json
```

Don't point live open-relay traffic at a fresh SignalMTA install to see what happens. Lock down relay sources first, then test queue admission using sources you've already authenticated. If the spool, DNS resolver, TLS policy, or domain authentication isn't ready, SignalMTA needs to reject or defer that mail safely rather than accept something it can't protect.

Evaluation Guide

This guide applies to the 0.1.2-rc4 release. SignalMTA is customer-operated software: install it on a Linux host you control, apply the issued evaluation license, and validate the exact environment you intend to operate.

What the public demo represents

The public demo uses the same management-console source and the same CTO, Engineer, and Deliverability views shipped in the customer package. It is intentionally different in four ways:

- It uses representative operational data rather than customer telemetry.
- It creates an anonymous, read-only session only for the configured public demo hostname.
- It disables state-changing actions.
- It adds the public-demo banner and six-step tour.

A customer installation requires a bootstrapped administrator, OTP-protected authentication, a valid license, and explicit authorization for every operational change.

Standard trial

- 30-day Zeta evaluation
- Maximum two active instances
- Signed, entitlement-bound download
- Smart-sink validation before controlled live traffic
- No credit card
- Support through hello@signalmta.com

1. Verify the package

Keep the original archive, manifest, checksum, and release signature together. Confirm that they all identify the same version before installation.

```
sha256sum -c SHA256SUMS
tar -xzf signalmta-customer-0.1.2-rc4.tar.gz
cd signalmta-customer-0.1.2-rc4
cat VERSION
npm run version:check
```

Expected version: 0.1.2-rc4.

2. Prepare a controlled host

Use a current supported Linux distribution, local NVMe-backed spool storage, synchronized time, a reliable caching resolver, and explicit firewall rules. Confirm outbound port 25 policy before planning direct-to-MX delivery.

Do not expose a fresh SMTP listener as an open relay. Start with loopback, private CIDRs, authenticated submission, or another explicitly authorized source.

3. Install and bootstrap access

Follow the Docker Compose quickstart or native systemd installation guide included in the package. Create the first real administrator before exposing the console.

```
SIGNALMTA_BOOTSTRAP_PASSWORD='use-a-long-unique-password' \
  ./bin/signalmta bootstrap-admin \
  --config /etc/signalmta/signalmta.json \
  --in-place \
  --email ops@example.com \
  --name 'Ops Admin' \
  --replace-existing
```

Store the one-time TOTP setup material and recovery codes in the evaluator's password vault. Confirm that the console requires authentication and that demo mode is disabled.

4. Apply and activate the license

```
./bin/signalmta license apply --key "$SIGNALMTA_LICENSE_KEY"
./bin/signalmta license status
```

Confirm the Zeta edition, expiry date, organization, active node, and two-instance limit. A third simultaneous activation must be rejected.

5. Establish safe submission

Configure explicit relay allowlists, SMTP AUTH, API tokens, or mTLS. Submit small test messages through both intended intake paths. Validate:

- SMTP/API acceptance is measured separately from remote delivery.
- Accepted recipients produce durable queue writes.
- Unauthorized relay attempts are rejected before admission.
- Backpressure is visible and does not silently drop accepted mail.
- Message, trace, tenant, stream, route, and recipient identifiers remain correlated.

6. Configure identity and delivery

Create a sending domain and confirm SPF, DKIM, DMARC, return-path/VERP, rDNS, and TLS posture. Configure the intended tenant, stream, route, source IP, IP pool, throttles, retry policy, bounce

handling, complaints, suppressions, webhooks, and exports.

Keep delivery in smart-sink mode until the configuration and operational evidence are complete.

7. Validate the WebUI

The same packaged console must render correctly in all three modes.

- CTO mode - start with Overview and Configuration. Confirm health, capacity, license footprint, infrastructure utilization, cost, and risk are understandable without operational write controls.
- Engineer mode - start with Overview, Queue Monitor, Delivery Attempts, Traffic Shaping, IP Pools, Routes, Configuration, and Users. Confirm queue evidence is readable, long values remain contained, filters preserve scope, authenticated changes validate before publication, and rollback/audit evidence is available.
- Deliverability mode - start with Destination Sites, Sending Domains, Bounces & FBLs, Queue Monitor, Warm-up, SignalAI, and the Live SMTP Log. Confirm provider behavior, DSN strings, authentication, affected streams, source IPs, retries, complaints, and recommended actions remain connected.

At a minimum, test the console at a 1440 x 1000 desktop viewport and one mobile viewport. Wide evidence tables may scroll inside their panels; the application itself must not create unbounded horizontal overflow.

The WebUI is for configuration management and the Live SMTP Log. Do not validate native historical reports — comprehensive reporting is provided by BigStats.email, purchased separately.

8. Validate metrics and telemetry export

```
./bin/signalmta metrics
./bin/signalmta metrics --json
```

Confirm current runtime metrics are returned, then configure a signed HTTP POST webhook to BigStats.email or to your own HTTPS endpoint and verify that delivery events arrive and their signatures validate.

9. Run qualification checks

```
npm run smoke:smtp
npm run audit:security
npm run audit:trial
npm run check:dns-pressure
npm run check:recovery
npm run audit:golive
npm run audit:throughput
npm run audit:performance
npm run support:bundle
```

Record the host profile, configuration revision, build version, artifact checksum, message size, concurrency, DNS behavior, queue-write results, sink delivery, delivery attempts, DSN evidence, restart recovery, and support-bundle location.

10. Controlled live delivery

Enable public SMTP delivery only after relay security, sender authentication, complaint/unsubscribe processing, suppression behavior, queue integrity, DNS, TLS, retry, and logging checks pass.

Start with engaged recipients and conservative per-domain caps. Stop expansion for unexpected queue growth, authentication failure, elevated complaints, unexplained delivery attempts, or source-IP mismatch.

Acceptance checklist

- Signed package, manifest, checksum, and version agree.
- Administrator login and OTP work; anonymous customer-console access is denied.
- License applies and the instance limit is enforced.
- SMTP and API submission use approved credentials or sources.
- Accepted recipients are written durably and survive restart.
- Smart-sink delivery, delivery attempts, retry timing, and DSN evidence are visible.
- All CTO, Engineer, and Deliverability views render and navigate correctly.
- Configuration workflows and the Live SMTP Log work as expected.
- ./bin/signalmta metrics (and --json) returns current runtime metrics.
- Signed webhook export reaches BigStats.email or a customer-owned HTTPS receiver.
- Domain authentication and source-IP routing are verified.
- Bounces, complaints, unsubscribes, suppressions, webhooks, and exports are tested.
- Go-live preflight passes before controlled public delivery.
- A redacted support bundle is retained.

For help, email hello@signalmta.com with the version, checksum, deployment method, failing step, and a redacted support bundle. Never send private keys, passwords, bearer tokens, full license keys, or message bodies through ordinary email.

Core Concepts and Feature Definitions

Think of this section as your plain-language glossary for SignalMTA. You'll run into these same terms all over the WebUI, API, CLI, config files, logs, reports, and the Signal Scripting Language examples.

Adaptive Delivery Policy — A scoped rule that watches live delivery evidence and adjusts behavior within limits you approve. It can lower a domain/MX/source-IP/pool rate, extend retry timing, freeze a warmup grouping, or hold a stream for review when evidence crosses a threshold. — It lets SignalMTA react to provider throttling, reputation changes, and queue pressure without forcing operators to manually chase every 4xx spike. Guardrails keep automation narrow and auditable.

Traffic shaping — Delivery limits applied by recipient domain, MX rollup, tenant, stream, route, pool, source IP, connector, priority, or error category. — Shaping keeps one provider, customer, campaign, or IP problem from consuming the whole MTA.

Throttling — A temporary or configured reduction in sending rate, connection count, or retry pace. — Throttling is the safe response when a mailbox provider slows acceptance, a pool is saturated, or a source IP needs cooldown time.

Retry policy — The schedule and rules used after a temporary failure. Retry policy can vary by error class, provider, stream, pool, source IP, and priority. — Greylisting, rate limits, TLS policy failures, and block-oriented replies need different retry timing and operator actions.

Queue — Accepted recipients waiting for first delivery, retry, hold release, expiration, or final handling. — Queue age, depth, priority, and state tell operators whether mail is moving normally or blocked by policy, provider behavior, route limits, or worker capacity.

Spool — Durable on-disk storage for accepted message bodies, metadata, route decisions, retry state, and delivery evidence. — The spool is the source of truth for accepted mail. RAM accelerates scheduling and counters, but the spool protects queue integrity.

Queue age — How long a recipient has been waiting in a specific queue state or scope. — A small old queue can be more urgent than a large new queue because it may indicate a provider block, broken route, exhausted warmup cap, or stalled worker.

Priority reserve — A configured share of scheduler capacity reserved for high-priority traffic, usually transactional or customer-critical mail. — Bulk mail should not starve password resets, invoices, alerts, or other urgent streams.

Scheduler candidate window — The bounded set of ready recipients examined by the scheduler during a delivery cycle. — Candidate windows let SignalMTA make fast, fair selection decisions without scanning an entire high-volume spool on every tick.

Tenant — A customer, business unit, brand, or logical owner of traffic. — Tenancy keeps permissions, reports, pools, suppressions, webhooks, and delivery evidence separated.

Stream — A traffic class inside a tenant, such as transactional, marketing, lifecycle, alerts, or product updates. — Streams let operators route, throttle, warm up, report, and suppress different mail types differently.

Route — The rule that decides where a submitted message goes: virtual instance, queue lane, source-IP pool, specific IP, proxy, relay connector, DKIM identity, TLS policy, fallback behavior, and

priority. — Routes prevent a sender from accidentally using the wrong pool, source IP, identity, or provider connector.

Virtual instance — A logical MTA identity inside one installation. It groups routes, pools, limits, reporting, and permissions without requiring a separate Linux server. — Virtual instances help service providers and large senders separate customers or use cases while still operating shared infrastructure.

IP pool — A named group of source IPs with shared routing, rate limits, warmup state, tenancy, reporting, and reputation controls. — Pools separate shared, dedicated, warmup, cooldown, transactional, marketing, regional, proxy-owned, and provider-specific traffic.

Dedicated IP — A source IP or pool reserved for one tenant, brand, stream, or use case. — Dedicated IPs isolate reputation and reporting for service-provider and customer-specific use cases.

Shared pool — A pool used by multiple approved tenants or streams. — Shared pools improve utilization but require strict stream permissions, complaint monitoring, and fair scheduling.

Warmup pool — A pool containing new or reputation-unknown source IPs under controlled volume growth. — New IPs should not be used at full volume immediately. Warmup protects reputation and reveals provider acceptance behavior gradually.

Warm overflow pool — An already-established pool that can receive eligible mail when a warmup cap is exhausted. — Overflow prevents warmup limits from delaying time-sensitive mail while still protecting the new IP.

Cooldown — A temporary reduction, hold, or pause applied after risk signals such as throttling, block-oriented replies, complaint spikes, or proxy/source-IP mismatch. — Cooldown contains reputation risk and gives operators time to verify the cause.

MX rollup — A grouping of recipient domains that resolve to the same provider family or MX infrastructure. — Provider behavior often follows MX infrastructure rather than only the visible recipient domain.

Destination provider — The mailbox provider, corporate mail gateway, ISP, or hosted MX family receiving mail. — Delivery problems are often destination-provider-specific, so operators need provider-aware reporting and shaping.

Source-IP tenancy — The ownership and permitted-use model for an IP: shared, dedicated, leased, proxy-owned, regional, warmup-only, cooldown, or reserved. — Tenancy labels keep routing, reporting, suppression, and accountability clear.

Outbound proxy / proxy-owned egress — A network layer such as HAProxy, custom egress service, Linux policy routing, or Kubernetes egress gateway that owns the final source address while SignalMTA owns the delivery decision. — Proxy-oriented IP management lets operators control large IP inventories without binding every public IP directly to the MTA process.

Connector-backed route — A route that sends selected mail through an external delivery service or relay provider instead of native SMTP delivery. — Some customers need SendGrid, Mailgun, SparkPost, or another relay for specific mail classes while keeping SignalMTA as the policy and reporting control point.

API injection — Submitting messages to SignalMTA through the HTTP API rather than SMTP. — API injection is useful for application integrations, idempotency, structured metadata, and controlled backpressure.

SMTP submission — Submitting messages through SMTP listeners such as port 587 with STARTTLS/authentication, port 25 for controlled relay, or a custom internal port. — Secure submission controls prevent open relay exposure and preserve stream attribution.

X-Signal headers — SignalMTA-owned headers that can request or carry approved routing, stream, tenant, priority, pool, tracking, or policy hints. — They give sophisticated senders controlled influence without bypassing authorization or route policy.

Custom X-headers — Customer-supplied headers used for campaign, tenant, trace, or application reporting. — Preserving approved custom headers helps customers join SignalMTA logs with their own systems.

List-Unsubscribe — RFC-defined headers that tell mailbox providers and mail clients how a recipient can unsubscribe, including one-click unsubscribe when configured. — Proper unsubscribe handling reduces complaints, supports compliance, and improves mailbox-provider trust.

VERP — Variable envelope return path, where each message or recipient uses a unique bounce address that encodes correlation data. — VERP connects asynchronous bounces back to the original recipient, stream, source IP, route, and attempt history.

Out-of-band bounce — A DSN or returned message received after the original SMTP transaction was accepted. — Without correlation, operators may not know which message, route, or source IP caused the failure.

Bounce intelligence — Classification that combines SMTP codes, enhanced status codes, raw text, localized diagnostics, provider profiles, delivery history, and stream context into human-readable categories. — Operators need to know what is happening, not just see 550 or 421. Categories guide retry, suppression, throttling, hold, or deliverability review.

Feedback loop / FBL — A mailbox-provider complaint feed, usually ARF-formatted, that reports when a recipient marks mail as spam. — Complaints must feed suppression, reputation monitoring, source-IP reporting, and customer accountability.

Suppression — A rule that prevents future mail to a recipient, domain, stream, or scope after complaints, unsubscribes, hard bounces, policy decisions, or operator action. — Suppression protects recipients, sender reputation, and compliance configuration.

Webhook export — Signed HTTP delivery of events such as delivered, deferred, bounced, complaint, unsubscribe, policy, TLS, auth, queue, and DSN events. — Webhooks let customers feed dashboards, warehouses, billing systems, CRMs, and suppression processors.

Observability — The WebUI, logs, metrics, health checks, event exports, reports, and support bundles used to understand the system. — SignalMTA’s WebUI is the primary observability surface; Prometheus/Grafana and JSON exports are supported when customers need external monitoring.

Support bundle — A redacted diagnostic package containing configuration configuration, health, logs, queue summaries, version, OS facts, license state, and recent events. — Bundles reduce support time while avoiding exposure of message bodies, private keys, passwords, tokens, and full license keys.

Runtime lease — A local or cluster-visible record that a licensed node is actively using an allowed instance slot. — Leases protect purchased instance limits while allowing multi-instance keys for customers who licensed more than one node.

SignalAI advisory — An explainable recommendation based on queues, provider replies, bounces, DNS, TLS, auth, warmup, exports, cluster routing, and license configuration. — SignalAI helps operators move faster, but actions should remain evidence-backed and controlled by policy.

Signal Scripting Language hook — A deterministic policy hook invoked at defined points such as injection, route selection, pre-delivery, SMTP reply handling, bounce handling, or logging. — SSL gives advanced operators granular control without patching the MTA runtime. Hooks should be narrow, reviewed, and tested before live use.

Evaluation Paths

SignalMTA gives evaluators a structured start without taking the wheel away from experienced operators. The WebUI ships with an Evaluation page where you pick Guided setup, Expert mode, or Import existing config, and you can switch between them whenever you like.

How to read this guide

If you've run MTAs before, start with sizing, configuration, queueing, traffic shaping, authentication, logging/export, and performance qualification. If your team is setting up infrastructure for the first time, run Guided setup and the Environment Install Configurator first, then come back to the deeper sections once you're configuring live domains, IP pools, and delivery policy. Deliverability teams should focus on domains, bounce intelligence, feedback loops, reporting, source-IP strategy, and SignalAI recommendations.

Guided setup

Pick this when you want an ordered first-run sequence. SignalMTA walks you through license application, admin access, relay authorization, domain authentication, host qualification, webhook testing, and go-live checks in a sensible order.

Expert mode

Pick this when your team already knows its way around MTA operations. Every console, API, CLI, config, queue, routing, IP-pool, rate-limit, logging, and scripting control is available right away. Checks stay advisory unless something's actually blocking you.

Import existing config

Pick this when you're migrating off another MTA. Import your domains, source IPs, pools, routes, throttles, and delivery policies, then review the SignalMTA configuration it generates before you publish it.

Host qualification

Run the qualification suite to gather evidence on queue durability, DNS pressure, throughput readiness, crash recovery, and how logging/export is set up, and to run go-live checks, all before you send any real live traffic.

Expected defaults

You shouldn't have to prove basic MTA functionality exists. Out of the box, SignalMTA denies relay by default, ships disk-safe queue settings, caches DNS, applies sane TLS policy defaults, logs operations, supports WebUI/API/CLI/config workflows, and hot-reloads license changes.

What can still block operation

SignalMTA only blocks or heavily flags things that could actually hurt safety or live delivery: an open relay exposed to the internet, an invalid license entitlement, public SMTP mode running without explicit go-live confirmation when that guard is turned on, an unsafe queue/storage setup, an invalid configuration publish, or an instance count over what you've purchased.

Evidence expected before meaningful live traffic

Relay configuration — Only approved SMTP/API sources can inject mail; no open relay exposure exists. — Security, Evaluation, config validation.

Domain authentication — DKIM selectors, SPF alignment, DMARC policy, return-path domain, and TLS configuration are visible and testable. — Domains, Authentication, domain-auth plan.

Queue durability — Accepted mail survives process restart and disk recovery checks; queue state remains explainable. — Queues, host qualification report.

DNS pressure — Resolver latency, negative caching, MX lookups, and timeout behavior remain bounded under load. — DNS diagnostics, host qualification report.

Logging and exports — Connection logs, delivery-attempt logs, webhook signatures, export retries, and support bundles work before production incidents. — Logs and Exports, Observability.

Go-live configuration — Public SMTP mode is intentionally enabled only after safety, authentication, TLS, queue, and rate-limit checks pass. — Evaluation, go-live preflight.

```
./bin/signalmta evaluation status --config /etc/signalmta/signalmta.json
./bin/signalmta evaluation status --path guided --json --config /etc/signalmta/signalmta.json
./bin/signalmta qualify-host --config /etc/signalmta/signalmta.json \
  --report-dir /var/log/signalmta/readiness \
  --profile alpha-250k \
  --target-mph 250000 \
  --edition alpha
```

Environment Install Configurator

Run this configurator before you install anything. Pick the environment you're deploying into, the workload you're trying to qualify, and how your team likes to operate. Based on that, you'll get a practical install path, a starting resource profile, the WebUI tasks to do first, CLI/API commands to validate with, and, if you want it, an operating brief for an AI agent.

We keep this deliberately conservative and won't hand you a fixed throughput number, because real delivery capacity hinges on source-IP reputation, provider limits, DNS behavior, TLS negotiation, message size, how much logging/export you're generating, and the host itself. Treat the output as your install plan, then run your own qualification tests before you push live traffic through it.

- Build My Install Plan
- Environment Type
- Cloud-Hosted Linux VM
- Provider Or Facility
- Amazon Web Services (AWS)
- Intended Workload
- Evaluation And Functional Testing
- Primary Use Case
- Transactional/Triggered Mail
- Availability Target
- Single Node With Documented Recovery
- Operator Approach
- Mostly the WebUI
- Generate the Steps
- Reset Form
- Evaluation And Functional Testing On Cloud-Hosted Linux VM

If you're on AWS, don't confuse SES restrictions with self-hosted SMTP sending, file for any port-25 removal you'll need, plan around Elastic IP/rDNS where it applies, and pull CloudWatch cost and utilization numbers for CTO mode.

- Recommended Starting Resource Profile
- CPU
- 8 vCPUs
- Memory
- 16 GB of RAM

- Storage
- 250 GB of local NVMe or provisioned SSD
- Network
- 1 Gbps network, outbound SMTP policy confirmed
- Installation and Qualification Checklist

Make sure your cloud account allows outbound SMTP on the ports you actually need, and that you can control reverse DNS for every sending IP.

Pick compute-optimized or storage-optimized instances, and put the active spool on local NVMe or provisioned high-IOPS storage.

Put the MTA in a VPC/subnet where only approved injection sources can reach the SMTP/API listeners.

Install a supported LTS Linux distribution, turn on time sync, create the signalmta user, and bump up file descriptor and process limits.

Set up separate paths for /opt/signalmta, /etc/signalmta, /var/lib/signalmta/spool, /var/lib/signalmta/runtime, and /var/log/signalmta.

Install SignalMTA, bootstrap your first administrator account, turn on OTP, and log into the WebUI.

Apply the SignalMTA license key and check that the node/edition status shows active.

Add your first sending domain, work through the DKIM selector guidance, publish SPF/DMARC/return-path records, and check DNS resolution from more than one resolver.

Lock down who can relay: approved CIDRs, SMTP AUTH users, API tokens, mTLS if you need it, and per-source rate limits.

Set up your streams, queue priorities, routes, IP pools or dedicated IPs, traffic-shaping caps, retry schedules, and webhook exports.

Before you touch public delivery, run smart-sink tests, config validation, a completion audit, domain-auth checks, queue admission tests, and performance qualification.

Only flip on live SMTP delivery once authentication, bounce processing, logging, TLS, DNS, warmup, and rollback checks all pass.

Set up a high-priority transactional queue lane with worker capacity reserved just for it.

Use tight retry schedules, alert on queue age early, and keep this traffic on separate IP pools from your bulk marketing mail.

Only add List-Unsubscribe where the message class actually calls for it, and route OTP/password-reset mail through dedicated paths.

Write down your recovery plan: backup config, references to secrets, license state, runtime state, and how to pull a support bundle.

Keep a tested path for standing up a replacement host, and confirm your restore commands actually work before you rely on them in production.

In the WebUI, work through Settings -> License first, then Domains, Security, IP Pools, Routes, Traffic Shaping, Webhooks, and Users.

Run the WebUI validation diff before you publish each configuration revision.

Have an administrator run the qualification commands below before you turn on live delivery.

Validation commands

```
./bin/signalmta bootstrap-admin --config /etc/signalmta/signalmta.json
./bin/signalmta license apply --key "$SIGNALMTA_LICENSE_KEY"
./bin/signalmta domain-auth-plan --domain example.com --selector sig1 --bounce-domain bounces.example.com
./bin/signalmta config validate --config /etc/signalmta/signalmta.json
./bin/signalmta completion-audit --config /etc/signalmta/signalmta.json
./bin/signalmta performance-qualification --config /etc/signalmta/signalmta.json --out reports/performanc
```

Operational watchpoints

Start with smart-sink delivery so you can check intake, queueing, logging, and WebUI workflows, and confirm exports work, all without any risk to public delivery.

Leave domain and IP warmup off until authentication and bounce handling are both passing.

How to use the generated plan

Work through the steps in order and don't jump straight from installation to live sending. Prove out the host, spool, DNS resolver, TLS, DKIM, relay controls, bounce path, logging exports, and support bundle workflow first. If your team lives in the WebUI, follow the console steps and have an administrator run the listed validation commands as part of go-live approval. If your team automates everything, keep the same sequence but push configuration through API/CLI revisions instead.

AI-agent-assisted implementation guidance

If you're going the agentic route, treat your AI agent team as help with implementation, not as something you let run production unattended. Let agents read the public docs, generated install plans, non-secret config examples, and sanitized command output. But require a human to sign off before anyone applies licenses, changes relay authorization, rotates DKIM keys, adjusts live rate limits, turns on public SMTP delivery, or drains/releases production queues.

Hardware and Software Recommendations

Your real throughput ceiling depends on hardware, network, DNS, TLS, message size, your source-IP plan, how destination providers behave, how much overhead your logging/exports add, and your configuration. Treat these profiles as a starting point, then qualify your own environment with SignalMTA's test tools.

Recommended operating system baseline

Stick with a current LTS Linux distribution: Ubuntu LTS, Debian stable, Rocky Linux, AlmaLinux, or an RHEL-compatible enterprise build all work.

Supervise the production service with systemd, unless you're deploying through Kubernetes instead.

Run chrony or systemd-timesyncd and don't let clocks drift. DKIM, TLS, logs, and support bundles all rely on time being correct.

Put the active spool on XFS or ext4 over local NVMe. Steer clear of network filesystems for the active spool unless you've specifically qualified that storage path.

Raise file descriptor, process, and socket limits to match your worker count and connection strategy.

Evaluation VM — 8 vCPU, 16 GB RAM, 250 GB NVMe, 1 Gbps — Functional testing, WebUI review, domain-auth validation, smart-sink tests. — Disk latency, DNS cache hit rate, webhook export latency.

Single-node production — 16 vCPU, 32-64 GB RAM, 1 TB NVMe, 5 Gbps — Serious single-node workloads with live delivery and observability enabled. — Queue age, provider deferrals, source-IP reputation, log volume.

High-volume node — 32+ vCPU, 64-128 GB RAM, multiple NVMe volumes, 10 Gbps — High-volume senders with multiple IP pools, heavy exports, and strict queue targets. — DNS pressure, TLS handshake rate, export backlog, spool recovery time.

Zeta cluster — 3+ delivery nodes plus management database/API plane — Service providers, high availability, routing intelligence, and fleet visibility. — Node health, route stickiness, policy distribution, per-node spool capacity.

Bare metal guidance

Bare metal gets you the most predictable storage and network behavior. Go with enterprise NVMe drives that have power-loss protection, split OS, log, and spool onto separate volumes where you can, add redundant power and network paths, and tune NIC queues for heavy connection churn. If you're running customer-critical streams, keep a spare node or restore target ready to go.

Cloud-hosted guidance

Cloud VMs are quicker to spin up and replace, but you still need to check port 25 policy, outbound bandwidth, sustained IOPS, noisy-neighbor effects, whether you can control reverse DNS, IPv4

sourcing, and how snapshots/backups affect performance. Go with compute-optimized or storage-optimized instances and put the active spool on local NVMe. Don't assume a high vCPU count buys you high SMTP throughput; DNS, TLS, and remote provider limits usually hit the ceiling first.

Small VPS evaluation profile

A small VPS works fine as your first evaluation environment if all you need is a working WebUI review, installer validation, license application, a walkthrough of DKIM/domain-auth, controlled SMTP submission, small live-delivery checks, and practice pulling a support bundle. Treat it as evaluation only, not proof of final throughput. Keep public access tight, turn on WebUI OTP, use a dedicated hostname just for evaluation, set up reverse DNS if this host will actually send mail, and keep rate limits conservative.

Use the evaluation host to walk through the whole operational flow: install SignalMTA with Docker or systemd, apply an evaluator license, bootstrap your first administrator, set up one sending domain and return-path domain, check DKIM/SPF/DMARC/TLS, lock down relay authorization, submit test mail through authenticated 587 or a private 2525 listener, run go-live preflight, pull a support bundle, and send just a small controlled live-delivery sample.

Don't lean on a small VPS to back up broad throughput claims. If you need real evidence of sustained performance, run the throughput, crash-recovery, and DNS-pressure suites on the same class of hardware a serious evaluator would actually use.

- DNS Pressure
- High Impact
- Remote Limits
- High Impact
- Spool Latency
- Medium Impact
- CPU Load
- Variable Impact

Installation, Layout, and Upgrades

A clean install keeps software, configuration, spool data, logs, and secrets in separate places. That split is what makes upgrades safer and turns a support bundle into something you can actually read instead of a mess to untangle.

/opt/signalmta — Versioned software releases. — Keep old releases until rollback windows expire.

/etc/signalmta/signalmta.json — Main runtime configuration. — Keep this in version control with secrets referenced, not embedded.

/var/lib/signalmta/spool — Durable queue and message metadata. — Use local NVMe and monitor free space aggressively.

/var/log/signalmta — Connection, attempt, audit, export, and diagnostic logs. — Size for bursty logging during incidents.

/var/lib/signalmta/runtime — Runtime state, cache, lease state, warmup state. — Back up state needed for management continuity.

Pick the install path that fits

Not every operator installs SignalMTA the same way, so we support more than one path. Spinning up a first evaluation? Use guided Docker Compose. Running bare metal or a high-performance cloud node? Go native with systemd. Building repeatable infrastructure? Drive it with unattended install inputs and keep the resulting configuration under change control.

Guided Docker Compose — First evaluation host, WebUI demo, evaluator walkthrough, fast rollback. — MTA runtime, management console, persistent spool volume, runtime state, logs, default deny relay controls, and health checks.

Native systemd — Bare metal, high-performance cloud nodes, operators who want direct OS control. — Dedicated service user, service units, local spool, log directories, file limits, runtime state, and preflight checks.

Unattended install — Repeatable cloud images, Terraform/Ansible/Packer, service-provider rollouts. — Noninteractive config from an answer file, license placeholder, admin bootstrap, network listeners, and validation report.

Kubernetes — Advanced Zeta users with existing container orchestration. — Stateful delivery pods, persistent spool volumes, management API/console services, Secrets, ConfigMaps, and readiness probes.

What to have ready before you install

The node's hostname, and the console's hostname too if you're running one.

Your license key, or an evaluator license file.

The initial administrator's email address and whether OTP is required.

Which submission listeners you need: usually authenticated port 587, and maybe controlled port 25 or a private 2525.

The relay CIDRs you're approving, plus SMTP AUTH users, API tokens, or mTLS client identities.

Where the spool lives and how much disk you're budgeting for it.

Your sending domain, your bounce/return-path domain, and a plan for DKIM selectors.

Your source-IP model: direct host IPs, egress owned by a proxy, leased IP blocks, or connector-backed routes.

Whether this node stands alone or is joining a Zeta cluster.

tar -xzf signalmta-<version>-linux-amd64.tar.gz

```
cd signalmta
sudo ./scripts/install-linux.sh --mode docker --guided --hostname mta-eval.example.com

sudo ./scripts/install-linux.sh --mode docker --answers ./install.answers.json --dry-run
sudo ./scripts/install-linux.sh --mode docker --answers ./install.answers.json --apply
```

Once the packaged installer finishes, you should have the MTA and console services running, persistent spool/runtime/log volumes in place, a first administrator locked behind OTP, relay denied by default, and a WebUI preflight page reporting on remaining license, relay, DNS, TLS, domain-auth, and queue checks.

Basic Installation

If you'd rather skip the manual setup work on your first host, ask for Basic Installation. It's a one-time paid install for a single Linux host, carried out by a SignalMTA AI-assisted installation agent with our team supervising. It sets up the SignalMTA runtime, the included WebUI, HTTPS, a persistent spool, runtime state, logs, first-admin bootstrap, and applies your license key.

This is a separate $500 one-time purchase. If SignalMTA already gave you an install coupon code, drop it into the Basic Installation checkout area before you head to Stripe so it's prefilled when you check out. Once you've paid, fill out the installation details form on the SignalMTA site so we can confirm the target hostname, Linux OS, license or evaluator reference, initial relay CIDRs, your preferred install window, and temporary agent access. This isn't a screenshare session — the agent runs the install non-interactively once you've approved temporary, revocable access.

The assisted installer comes up in smart-sink mode first. From that state, SignalMTA can accept authorized test submissions, run messages through queue admission, write logs, and show you evidence in the WebUI, all without sending any real outbound SMTP. Public SMTP and submission listeners stay bound to localhost until you explicitly turn live SMTP on, and only after relay authorization, DNS, DKIM/SPF/DMARC, TLS, queue, bounce, and security checks all pass.

```
sudo SIGNALMTA_RELEASE_URL="https://downloads.signalmta.com/dl/..." \
  SIGNALMTA_RELEASE_SHA256="expected-release-sha256" \
  SIGNALMTA_CONSOLE_DOMAIN="mta-console.customer.example" \
```

```
SIGNALMTA_ADMIN_EMAIL="ops@customer.example" \
SIGNALMTA_LICENSE_KEY="SIG-..." \
SIGNALMTA_RELAY_CIDRS="203.0.113.0/24,198.51.100.10/32" \
./scripts/signalmta-basic-install.sh
```

Before you request this service, get the target Linux host ready: a DNS hostname for the WebUI, a public IP, a supported OS version, your license or evaluator reference, initial relay CIDRs, a preferred install window, and a way to grant temporary access — a temporary SSH key, a temporary sudo user, or a customer-run automation runner with temporary SSH access. Don't send permanent passwords, API keys, private keys, or customer message data through the request form.

Once the agent-run install wraps up, SignalMTA emails a completion report to your contact. It covers the installed build, host and WebUI URL, service status, which listeners are enabled, the persistent paths, license status, preflight results, whether you're in smart-sink or live-SMTP mode, a sanitized summary of the install log, and what's still left for you to do. Secrets, private keys, OTP seeds, and reusable credentials never show up in that email.

After the assisted install is done, sign in to the WebUI, save the first-admin OTP and recovery material somewhere secure, run through Setup -> Preflight, set up your sending domains and return-path, confirm DKIM/SPF/DMARC/TLS are verified, add your IP pools and egress sources, send some smart-sink tests, check the logs and queue evidence, and only flip on live SMTP once the readiness checks are all green.

Native systemd installs run through the same preflight checks, but the runtime stays directly under the operating system:

```
sudo useradd --system --home /var/lib/signalmta --shell /usr/sbin/nologin signalmta
sudo install -d -o signalmta -g signalmta /var/lib/signalmta/spool /var/lib/signalmta/runtime /var/log/si
sudo install -d -o root -g signalmta /etc/signalmta
sudo systemctl enable --now signalmta
sudo systemctl enable --now signalmta-console
```

After you install, open the WebUI and run Setup -> Preflight. Don't go live until it's green across the board: license active, relay locked down, domain authentication verified, DNS resolver healthy, TLS setup confirmed, spool writable, log/export path writable, and at least one route mapped to an approved source-IP path.

Upgrade procedure

Run signalmta update-check and read through any safety blockers it flags.

Grab an export of the active configuration revision and save a support bundle alongside it.

Check that queue age looks acceptable and there's no emergency hold in progress.

Drop the new release into a versioned path under /opt/signalmta.

Run config validation and spool maintenance in check-only mode first.

Reload services if the changes are compatible, or do a supervised restart during your maintenance window.

Confirm SMTP submission, delivery attempts, WebUI login, metrics, webhooks, and queue recovery all work.

Versioning, Release Notes, and Build Verification

Every SignalMTA release ships with a visible version number, a software changelog, and a release manifest, so you can confirm exactly what build is running. Before promoting a build, the version should match everywhere — WebUI, CLI, package metadata, and release manifest.

The current release-candidate line is 0.1.2-rc1. Before you turn on any meaningful live traffic on a build, double-check relay authorization, DNS, source-IP, authentication, and queue controls.

```
./bin/signalmta version
./bin/signalmta native-worker version
./bin/signalmta native-worker build-check --config /etc/signalmta/signalmta.json
npm run package:customer -- --dry-run
```

VERSION — Single source for the release number packaged with customer builds. — Confirm it matches package.json and the WebUI footer/version view.

CHANGELOG.md — Human-readable release history for operators, evaluators, and support. — Read before updating, especially for delivery, queue, authentication, and config changes.

release-manifest.json — Machine-readable package manifest produced during customer release packaging. — Verify channel, version, required files, checksums, and native worker metadata.

signalmta-worker --version — Shows Go delivery engine build metadata and runtime contract version. — Run during non-production checks and attach output to support bundles when requested.

Before you install or upgrade, run version checks, package validation, native worker verification, a WebUI workflow audit, terminology linting, and a smart-sink delivery test. After the install, that same version number should show up in the WebUI, CLI, and release manifest.

Software Changelog

The SignalMTA changelog only tracks software changes: runtime, WebUI, API, CLI, packaging, delivery behavior, security controls, and compatibility notes.

View 0.1.2-rc4 Changelog

View 0.1.2-rc3 Changelog

View 0.1.2-rc2 Changelog

License Activation

License keys for SignalMTA come from SignalMTA sales or support. You apply the key inside your own customer-hosted console, CLI, or API. When it's safe to do so, the runtime picks up the new entitlement state on the fly, without knocking any delivery workers offline.

```
./bin/signalmta license apply --key "$SIGNALMTA_LICENSE_KEY"
./bin/signalmta license status --json
curl -X POST http://127.0.0.1:4780/api/license/apply \
  -H "Authorization: Bearer $SIGNALMTA_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"key":"SIG-..."}'
```

The WebUI lays out your license edition, active nodes, expiry warnings, which feature areas are enabled, and reload state. Put license checks on your operational dashboard so a key expiring never catches the team off guard mid-sending-window.

Configuration Format and Best Practices

SignalMTA stores configuration as structured JSON, and you can get at it however you like: edit it directly, drive it through the WebUI, hit the API, or script it with repeatable CLI workflows. No matter which path you pick, everything runs through the same validation, revision history, safety checks, hot-reload plan, audit trail, and rollback mechanism.

Don't think of the WebUI as its own separate config system, because it isn't one. When you change a domain, IP pool, route, webhook, TLS policy, or traffic-shaping rule there, SignalMTA turns that into a typed configuration patch, validates it, shows you a diff, logs who made the change and why, and publishes a fresh runtime revision.

```
{
  "node": { "name": "mta-01", "edition": "zeta", "spool_path": "/var/lib/signalmta/spool" },
  "smtp_receive": {
    "listen": [":25", ":587"],
    "max_message_bytes": 52428800,
    "relay": {
      "default": "deny",
      "allowed_cidrs": ["203.0.113.0/24"],
      "require_auth": true,
      "require_tls_for_auth": true
    }
  },
  "dns": { "cache_ttl_seconds": 300, "negative_ttl_seconds": 60, "max_concurrent_lookups": 2000 },
  "queues": {
    "spool_high_watermark_pct": 82,
    "transactional_reserve_pct": 25,
    "max_ready_per_domain": 250000,
    "fail_after_attempts": 12
  },
  "observability": {
    "prometheus": { "enabled": true, "listen": "127.0.0.1:9090" },
    "events": [{ "name": "delivery-json", "type": "jsonl", "path": "/var/log/signalmta/delivery.jsonl" }]
  }
}
```

Format rules

JSON object — Keep the root as an object with named sections such as smtp_receive, dns, queues, ipPools, authentication, and observability. — Allows deterministic validation, diffing, and API/WebUI patch generation.

No embedded secrets — Use secret references such as env://SIGNALMTA_WEBHOOK_SECRET, file:///etc/signalmta/secrets/dkim.key, or managed secret providers. — Prevents support bundles, config diffs, and Git history from exposing credentials.

Explicit units — Use clear unit-bearing names such as maxMessageSizeMb, dnsCacheTtlMs, retryWindowSeconds, or string durations such as 30m. — Reduces mistakes when tuning performance and retry behavior.

Stable IDs — Give pools, routes, virtual instances, streams, webhooks, and warmup plans stable IDs. — Logs and reports can join events to the same object across revisions.

Environment overlays — Use separate files or patches for lab, evaluation, and production values. — Keeps production limits and source IPs from leaking into test environments.

Validated publishes — Always validate before publishing, even for WebUI changes. — Blocks open relay controls, missing DKIM, unsafe TLS, bad routes, and unsafe spool limits.

Recommended file layout

```
/etc/signalmta/
  signalmta.json              # active production config
  conf.d/
    10-node.json              # node identity, paths, listener binding
    20-security.json          # relay policy, auth, users, RBAC references
    30-domains.json           # sending domains, DKIM, TLS, return-path
    40-routing.json           # virtual instances, IP pools, routes
    50-shaping.json           # recipient-domain/MX caps, retry, warmup
    60-observability.json     # logs, webhooks, Prometheus, export
  ssl/
    injection.ssl
    smtp-reply.ssl
  secrets/                    # root-readable references, not committed
```

Publishing config from the WebUI

Head into Settings and pick the area you want to touch: Domains, Routes, IP Pools, Traffic Shaping, Webhooks, Security, Users, or Cluster.

Make your change with the guided form controls. You'll see required fields, allowed ranges, route permissions, and any safety warnings right there inline.

Hit Validate. SignalMTA checks the syntax, the references, whether secrets are available, domain authentication, how open your relay is set up, the impact on the queue, and whether the change is hot-reload compatible.

Look over the diff it generates along with the reload plan. The WebUI tells you whether the change can hot-reload, needs a worker drain first, or should be scheduled for later.

Type in a reason and click Publish. SignalMTA cuts a new revision, logs who did it, and rolls out the runtime update safely.

If the new policy doesn't behave the way you expected, pull up the revision history and use Rollback.

Publish workflow

```
./bin/signalmta config validate --config ./candidate.json
./bin/signalmta config diff --from active --to ./candidate.json
./bin/signalmta config publish --config ./candidate.json --reason "Add new warmup pool"
./bin/signalmta config rollback --to previous
```

These commands run the exact same validation and publish steps the WebUI uses under the hood. Reach for the CLI when you want repeatable automation, source-controlled change management,

support diagnostics, or you'd just rather work from a terminal. You don't need it at all if you're happy running everything from the WebUI day to day.

During validation, SignalMTA refuses to let dangerous configuration through: an open relay setup, public listeners with no authentication, DKIM references that don't resolve, public streams with no bounce domain, spool limits that aren't safe, required TLS routes missing their TLS material, and unlimited delivery rates on domains that are already seeing deferrals.

WebUI Functionality

What the WebUI is for

In 0.1.2-rc4 the included WebUI is dedicated to configuration management and real-time SMTP operational visibility. Historical reporting, trend analysis, bounce and deferral analysis, source-IP performance, provider/domain reporting, and long-window analytics are handled by BigStats.email, which is purchased separately from the SignalMTA license.

History & Reports has been removed from the default navigation, and the console no longer loads native reports. The Live SMTP Log is the operational visibility surface: watch connections, delivery attempts, and SMTP replies as mail is processed.

The included WebUI is your operating console for the MTA. From it you can watch queues, delivery attempts, domains, IP pools, routes, bounces, complaints, warmup, SignalAI advisories, configuration revisions, users, security settings, logs, and exports.

SignalMTA dashboard screenshot

Dashboard: a quick read on delivery, queue health, domain trends, and how things are running overall.

WebUI modes

CTO — Technical executives and infrastructure owners. — Fleet summary, license footprint, delivery health, cloud utilization, estimated CPM, risk summary, and SignalAI executive advisories.

Engineer — MTA operators and platform admins. — Queues, workers, source IPs, pools, routing, TLS, DNS, service health, config revisions, logs, and operational controls.

Deliverability — Deliverability specialists and email operations teams. — Domain drilldown, stream analysis, bounce categories, complaint trends, warmup state, provider issues, SignalAI recommendations.

WebUI operator handbook

The WebUI is built so you can handle day-to-day admin work without ever touching a terminal. Every form writes into the same runtime configuration model the CLI and API use, and every publish gives you validation output, a diff, a reload plan, an audit record, and a rollback point.

Navigation model

Dashboard — Confirm whether the platform is healthy right now. — CTO, Engineer, Deliverability

Queues — Understand why mail is waiting, retrying, held, or permanently failed. — Engineer

Domains — Add sending domains, verify DNS, inspect DKIM/TLS/authentication evidence. — Engineer, Deliverability

Traffic Shaping — Control rates by domain, MX, stream, pool, source IP, connector, or error class. — Engineer, Deliverability

Routes and IP Pools — Decide which customer, stream, domain, or campaign can use which pool, IP, proxy, or relay connector. — Engineer

Warmup — Create and monitor automated SignalWarmup plans for new IPs. — Deliverability, Engineer

Reports — Explain delivery outcomes by customer stream, domain, campaign, pool, source IP, and category. — CTO, Deliverability

Logs and Exports — Trace individual messages, inspect webhooks, export JSON, and build support bundles. — Engineer, Deliverability

Security — Manage relay permissions, users, OTP, API tokens, SMTP AUTH, mTLS, and audit configuration. — Engineer

SignalAI — Review anomaly explanations, evidence, recommended action, and approval state. — CTO, Engineer, Deliverability

Publishing changes safely

Open the relevant area — Domains, Traffic Shaping, Routes, Warmup, Webhooks, Security, whatever applies.

Click Add, Edit, or Create rule to start the change.

Fill in the form fields. Required fields get validated before you can stage the change.

Click Validate. Fix anything that blocks you before moving on. Warnings tell you about risk but don't always stop publication.

Check the generated diff and make sure only the fields you meant to touch actually changed.

Read the reload plan. It tells you whether the change is hot-reloadable, needs a worker drain, or should be scheduled.

Enter an audit reason and hit Publish.

Check the result in the screen-specific verification area — authentication evidence, route match preview, queue behavior, delivery-attempt logs, or webhook health, depending on what you changed.

Don't publish a broad change to fix a narrow problem. Scope it down instead — by tenant, stream, domain, MX, pool, source IP, or provider grouping — so healthy traffic keeps flowing.

Dashboard operations

Open Dashboard at the start of your shift or when reviewing an incident. Use CTO mode for fleet, license, cost, and risk summaries; Engineer mode for queue age, worker state, DNS/TLS status, route health, and controls; and Deliverability mode for sending domains, recipient-domain/MX behavior, bounces, complaints, warmup, message streams, and SignalAI recommendations.

Check the trends for delivered, deferred, bounced, complaint, and retry.

Watch for any one metric moving out of step with the rest.

Click into the affected domain, stream, or pool to open the drilldown.

Compare the current window against the last window that looked healthy.

Open SignalAI for that same scope and look at the evidence before you touch policy.

Time windows and report scope

Every operational view needs a clear time scope. Dashboard, Queues, Domains, Reports, Logs, SignalAI, Webhooks, and CTO-mode cost views all share the same time-window model, so you can go from a summary chart straight to exact evidence without losing your place.

Reach for the global time picker when you want the whole WebUI answering the same question. Common presets: last 15 minutes, last 1 hour, last 2 hours, last 24 hours, last 7 days, last 14 days, last 30 days, last 60 days, or a custom absolute range. When you're chasing an incident, start with the smallest window that shows the symptom, then compare it to the last healthy stretch. For trend work, pull 14, 30, or 60 days and let reports roll noisy minute-level events up into hourly or daily buckets.

Recent windows use finer buckets, so short throttling events, webhook backlogs, TLS failures, and queue-age spikes still show up clearly.

Multi-week and multi-month windows use rollups instead, which keeps reports fast and readable.

Every chart, table, export, and support bundle keeps a record of the time range, timezone, filters, and rollup interval you picked.

Logs let you switch between event time and ingestion time. Use event time when you're analyzing delivery, and ingestion time when you're chasing export lag.

Drilldowns keep your scope intact. Filter Reports to last 2 hours, gmail.com, transactional, and pool-east, and the linked logs, queue samples, bounce categories, and SignalAI view all open with that same scope already applied.

In Reports, set the time window to last 2 hours.

Filter down to the affected tenant or stream.

Compare against the previous 2 hours.

Click into the affected domain, MX, pool, source IP, or bounce category.

Open the linked delivery-attempt logs, keeping the same time range.

If the problem goes back further, widen to last 24 hours or last 14 days to figure out whether this is a fresh incident or a pattern that keeps repeating.

Queue operations

Open Queues whenever mail isn't moving the way you expect. Filter by tenant, stream, domain, MX, pool, source IP, priority, and queue state. Sort by queue age, pull up a message sample, and check the route, retry history, SMTP replies, TLS evidence, and policy decisions before you act.

Ready — Eligible for delivery now. — Check priority, domain caps, worker capacity, and source-IP availability.

Deferred — Waiting for retry after a temporary result. — Inspect SMTP code, category, retry time, and shaping rule.

Held — Paused by operator, policy, warmup, or SignalAI advisory. — Release only after the hold reason is resolved.

Watched — Sampled for closer inspection because risk signals changed. — Review evidence before applying broad changes.

Permanent failure — Final failure after recipient, policy, or delivery classification. — Suppress, export, or review; do not replay unless classification is wrong.

Quarantine — Metadata/body integrity issue or unsafe artifact. — Run safe repair workflow before any redrive.

Domains and authentication

In Domains, click Add sending domain.

Provide the From domain and the return-path domain.

Choose standard DKIM, DKIM2, or double-signing if a service-provider customer needs two aligned signatures.

Generate or reference selector keys, then copy the DNS records exactly as shown.

Click Verify DNS and wait for selector, SPF, DMARC, return-path, and TLS checks to pass.

Bind the domain to the approved tenants, streams, routes, pools, or virtual instances.

Publish the revision, send a test message, and check the Authentication evidence.

If DNS fails, the WebUI shows you the hostname it queried, the resolver's answer, the parsed value, and a recommended fix. Hold off on scaling traffic until that authentication evidence comes back clean.

Routes, pools, dedicated IPs, and proxy egress

In IP Inventory, create or import the source IP.

Check reverse DNS, SPF alignment, abuse contact, and proxy mapping.

Name a new IP pool after the customer or traffic class.

Assign source IPs to it, and mark the pool as shared, dedicated, warmup, cooldown, provider-specific, or connector-backed.

Bind the pool to the customer's tenant and the streams it's allowed to serve.

Attach an automated SignalWarmup plan if the IP is new or its reputation is unknown.

Create a route rule that matches the customer stream and picks that pool.

Run Preview route match with sample headers before you publish.

Check the delivery-attempt logs to confirm the expected pool, source IP, proxy id, DKIM selector, and route id all show up.

Traffic shaping and retry control

Open Traffic Shaping when a provider is slowing your mail down, a pool is saturated, a source IP is cooling down, or a customer stream needs its own limits. Shape the smallest scope that actually explains the problem.

Filter down by affected domain, MX, pool, source IP, stream, route, connector, or error class.

Open Explain policy to see the active cap and the reasoning behind it.

Review the recent SMTP replies and bounce categories.

Lower only the scope that's actually affected, or just accept the SignalAI recommendation.

Set a cooldown window along with retry jitter.

Add an audit reason and publish.

Confirm queue age drops for the affected scope without slowing down your high-priority streams.

Automated IP warmup

Open Warmup to automate ramping up a new IP. SignalWarmup means you're not hand-editing daily caps yourself — you define the plan and guardrails, and the controller looks at the evidence and decides whether to increase, wait, freeze, or pause.

Click New warmup plan to get started.

Select the source IP, pool, proxy, tenant, and optional IPXO lease.

Pick a future activation date.

Enter the initial cap, target cap, maximum daily increase, and duration.

Choose recipient groupings — Gmail/Google MX, Microsoft consumer, Yahoo/AOL, Apple, Comcast, regional ISPs, B2B, or custom groups.

Point it at an already-warm overflow pool.

Set guardrails covering complaints, hard bounces, deferrals, block-oriented replies, authentication, reverse DNS, and TLS.

Publish the plan and keep an eye on the Controller's decisions.

Reports and deliverability drilldown

Select the tenant or customer stream.

Narrow it down by time window, campaign, tag, custom X-header, domain, MX, pool, source IP, route, or bounce category.

Check the trends for delivered, accepted, deferred, bounced, complaint, unsubscribe, and retry.

Open whichever domain or source IP is trending worst.

Compare SMTP replies, enhanced status codes, bounce categories, complaint rate, unsubscribe rate, TLS evidence, authentication evidence, warmup phase, and the SignalAI recommendation.

Export the filtered evidence whenever a customer, deliverability platform, or support case needs it.

Bounces, complaints, and unsubscribes

Narrow the filter to tenant, stream, campaign, domain, source IP, and category.

See whether the category is invalid recipient, remote throttling, block-oriented, spam-oriented, authentication, TLS, content policy, or regional/localized.

Open a sample of the evidence and check the raw text, UTF-8/localized diagnostics, enhanced status code, provider profile, and recommended action.

Suppress invalid recipients according to your policy.

Hold or slow the affected streams when you see block-oriented or spam-oriented signals.

Confirm webhooks and exports actually received the classification.

Logs, webhooks, and exports

Search using trace id, message id, idempotency key, tenant, stream, domain, MX, source IP, route id, proxy id, SMTP code, webhook event id, or a custom X-header.

Pull up the event timeline.

Confirm submission, queue admission, route selection, DNS lookup, TLS negotiation, DKIM signing, SMTP result, retry decision, bounce classification, and webhook export all happened as expected.

If a webhook is running late, check the pending count, oldest pending event, last response, signature status, and retry state.

Click Replay webhook only once you've confirmed the receiving endpoint is actually healthy.

Generate a redacted support bundle when a human support engineer needs the evidence.

Security, users, and OTP

Configure allowed relay CIDRs only for sources you trust.

Require SMTP AUTH, an API token, or mTLS for submission.

TLS has to be established before SMTP AUTH is allowed.

Set up named API tokens, scoped by tenant, stream, route, and permission.

Create users with a role and a default UI mode.

Turn on OTP enforcement for operator accounts.

Review failed-login throttling and the audit logs.

Double-check that no wildcard relay source is allowed.

SignalAI operations

Filter down to the same tenant, stream, domain, pool, source IP, or queue scope you're already investigating.

Review the severity, confidence, likely cause, supporting evidence, and recommended action.

Compare the recommendation against the raw logs and reports.

Approve guarded actions only once you've confirmed the affected scope is correct.

Review the generated config diff before you publish.

Confirm the outcome in queues, reports, and logs.

Common WebUI workflows

Finding a delivery blocker: open Deliverability Mode, filter by tenant/stream/domain, look at the bounce categories, compare pool and source IP performance, then check SignalAI's recommendations against the raw SMTP evidence.

Protecting transactional mail: open Engineer Mode and verify the priority lane reserve, queue age target, stream policy, and route assignment for your transactional traffic.

Investigating a domain: search for it, open its domain history, and look at MX-level throttles, deferrals, TLS results, DNS latency, and recent retries.

Reviewing a config change: open Settings, compare the revision diff, run the safety checks, publish, then watch the audit events and queue behavior afterward.

WebUI-only operation

You don't need the CLI to run SignalMTA day to day. The WebUI covers the full workflow on its own: apply license keys, create users, turn on OTP, create sending domains, generate DKIM records, configure return-path domains, create IP pools, assign dedicated IPs, configure warmup, publish traffic-shaping rules, inspect queues, hold or release traffic, review bounces, replay webhooks, and export support bundles.

Apply a license — Settings -> License -> Apply key — Validates the key, reloads entitlement state, and records an audit event.

Create a sending domain — Domains -> Add domain — Creates the domain-auth plan, DKIM selectors, return-path checks, and DNS validation tasks.

Create a dedicated IP pool — IP Pools -> New pool — Creates a pool ID, source IP assignments, tenant/stream permissions, warmup plan, and reporting labels.

Adjust traffic shaping — Traffic Shaping -> Domain or Pool policy — Creates a validated config patch with caps, adaptive triggers, retry behavior, and rollback metadata.

Investigate a bounce spike — Deliverability -> Bounces — Filters direct replies, asynchronous DSNs, complaints, unsubscribes, categories, and raw evidence.

Export logs — Observability -> Logs/Exports — Runs the same redaction, filtering, and bundle generation used by support commands.

SignalMTA queues screenshot

Queues: ready mail, deferred mail, held mail, permanent failures, priority, and how recovery is going.

WebUI Tour

The SignalMTA WebUI follows how delivery teams actually operate: check system state, narrow down the scope of a problem, look at the evidence, then ship a safe change. You can do the whole job from the WebUI, and the CLI and API expose those same actions for automation.

SignalMTA dashboard WebUI screenshot

This is a screenshot of the dashboard from a live SignalMTA WebUI.

Dashboard: live delivery status

The dashboard tells you at a glance whether mail is moving the way it should. Watch message statistics, runtime consumption, remote-accepted recipient totals, upcoming retries, per-domain delivery metrics, and overall health here. You can narrow the view by UI mode, active user, sending IP set, recipient domain, remote IP, or tag.

Check accepted, delivered, deferred, bounced, and retried volume.

Catch retry pressure building up, shifts in SMTP errors, and how the queue is moving.

Click a domain row to jump straight into troubleshooting for that domain.

Switch between CTO, Engineer, or Deliverability mode depending on what you're trying to do.

SignalMTA queues WebUI screenshot

This is a screenshot of queue operations from a live SignalMTA WebUI.

Queues: backlog, priority, and recovery

The Queues view lays out what's ready, deferred, held, watched, scanned, retryable, or terminal. Engineers use it to make sure high-priority transactional mail stays protected, and to decide whether traffic needs to be held, released, drained, throttled, or replayed.

You can filter this view by tenant, stream, route, domain, MX, pool, source IP, priority, and retry state.

See why a given message got deferred and when it's due to retry.

Hold or release traffic within a narrow scope instead of pausing the whole MTA.

Check permanent failures and the state of corrupt-message quarantine before you replay anything.

SignalMTA configuration WebUI screenshot

This is a screenshot of the configuration screen from a live SignalMTA WebUI.

Configuration: validated runtime changes

The Configuration area exposes the same runtime model backing the config files, CLI, and API. From here you manage sending domains, DKIM selectors, TLS policy, return-path domains, routes, virtual instances, IP pools, proxy egress, IPXO inventory, warmup plans, webhooks, users, OTP, API tokens, and cluster nodes.

Confirm domain authentication is set up correctly before you scale sending.

Build shared pools, dedicated IP pools, proxy-managed pools, and leased-IP assignments.

Look over diffs, publish revisions, see what a reload will affect, and roll back safely if needed.

Block publishes that would leave you exposed, like an open relay or a missing DKIM record.

SignalMTA traffic shaping WebUI screenshot

This is a screenshot of traffic shaping and policy from a live SignalMTA WebUI.

Traffic Shaping: delivery flow control

The Traffic Shaping view shows what policy is active right now for domains, MX hosts, pools, source IPs, streams, routes, connectors, and error classes. Use it to figure out why traffic slowed down, which adaptive rule fired, and whether a recommended adjustment is worth publishing.

Tune starting caps, connection limits, cooldown windows, retry schedules, and adaptive reductions here.

Look at throttling, block-oriented replies, spam-oriented replies, DNS pressure, and TLS-required failures.

Carve out reserved capacity for transactional or other high-priority streams.

Publish adjustments safely with an audit reason and a rollback target attached.

SignalMTA reports WebUI screenshot

This is a screenshot of the reports screen from a live SignalMTA WebUI.

Reports: delivery blockers and trends

Reports tie together delivery attempts, queue transitions, SMTP replies, bounce categories, complaint events, unsubscribe events, TLS evidence, DNS metrics, warmup stage, and SignalAI recommendations. Deliverability folks can drill into a customer stream and pin down whether the problem is the domain, the provider, the source IP, the campaign, authentication, or a content-response issue.

Check domain trends, source-IP pool performance, bounce categories, complaints, and export health.

You can filter by tenant, stream, campaign, tag, custom X-header, domain, MX, pool, or source IP.

Keep block, spam, throttling, reputation, policy, invalid-recipient, and transient categories separated out.

Pull evidence for deliverability platforms and support investigations.

SignalMTA logs and export WebUI screenshot

This is a screenshot of logs and export from a live SignalMTA WebUI.

Logs and Exports: reconstruct the message path

The Logs view lets you trace a message all the way from submission through queue admission, route choice, DNS lookup, TLS negotiation, DKIM signing, SMTP response, retry decision, bounce classification, webhook delivery, and report aggregation.

You can search by trace id, message id, idempotency key, tenant, stream, domain, MX, pool, source IP, proxy id, or event type.

Look at connection logs and delivery-attempt logs side by side.

Replay webhooks and check endpoint health, pending events, signatures, and the failed state for endpoints that ran out of retries.

Produce redacted support bundles that don't expose secrets or message bodies.

SignalMTA security WebUI screenshot

This is a screenshot of the security screen from a live SignalMTA WebUI.

Security: relay and operator control

The Security area covers how relay is locked down and how operator access works. Admins use it to keep the MTA deny-by-default, require authenticated submission, enforce OTP, scope API tokens, and audit sensitive changes.

Set up allowed relay CIDRs, SMTP AUTH users, mTLS clients, API tokens, and stream permissions.

Check OTP status, recovery codes, roles, UI mode assignment, and failed-login throttling.

The audit trail covers token creation, config publish, queue release, route change, DKIM rotation, and support bundles.

Watch the abuse-aware controls that keep bad injections from eating into queue capacity.

SignalMTA SignalAI WebUI screenshot

This is a screenshot of the SignalAI advisory view from a live SignalMTA WebUI.

SignalAI: recommendations with evidence

SignalAI watches delivery behavior and explains anomalies in plain operator terms. For each one it names the affected scope, the likely cause, its confidence, the supporting evidence, a suggested action, and whether a human needs to approve it.

It catches queue anomalies, provider deferrals, bounce spikes, DNS pressure, TLS/authentication issues, warmup drift, webhook backlog, and source-IP reputation changes.

Suggest actions like lowering a domain cap, pausing a pool, extending warmup, fixing DNS, or looking into a complaint spike.

Guarded actions, like route changes, pool holds, throttle adjustments, and suppression-impacting changes, still need a human to sign off.

Log every advisory, decision, actor, and resulting configuration change in the audit trail.

Message Submission

Mail gets into SignalMTA one of three ways: authenticated SMTP submission, HTTP API injection, or an approved connector. Don't think of submission as just a socket listener — it's the policy boundary. This is where SignalMTA figures out who's allowed to inject mail, which stream owns that traffic, which queue lane it lands in, which route/IP pool it can use, and whether it's even safe to accept into the durable spool.

Don't confuse these two metrics when you're troubleshooting. Remote accepted means the destination SMTP system took a recipient during delivery. Submission accepted means SignalMTA took the message into its own queue — a completely different event.

1. Submission ports and when to use them

You can point SignalMTA's listeners at any ports you like, but stick to conventional port roles so firewalls, load balancers, customers, and your own support staff can reason about the traffic path. Don't expose every listener to the public internet. Start from deny-by-default, then explicitly allow only the applications, customer networks, mTLS clients, and authenticated users that actually need to submit mail.

25/tcp — Public SMTP receiving or controlled MTA-to-MTA ingress. — Use only when SignalMTA is intentionally receiving mail from other MTAs or a tightly controlled internal relay layer. — Require relay policy. For customer submission, prefer AUTH/mTLS and strict source controls instead of open relay behavior.

587/tcp — Authenticated SMTP submission. — Recommended default for customer applications, ESP platform injectors, and internal senders. — Require STARTTLS before AUTH. Use SMTP AUTH, mTLS, or both. Reject unauthenticated relay attempts.

465/tcp — Implicit TLS submission when a customer stack requires it. — Optional compatibility listener. — TLS starts immediately. Require AUTH or mTLS before accepting mail.

2525/tcp — Local evaluation, private load balancer target, or alternate submission port. — Useful when cloud providers restrict port 25 or when testing behind a private proxy. — Keep private by default. If exposed, treat exactly like 587 with TLS and identity.

4780/tcp — Runtime API and WebUI in local development. — Put behind VPN, private network, reverse proxy, or management ingress in production. — Use authenticated API tokens, console login, OTP, and HTTPS termination.

On cloud-hosted deployments, check whether your provider even allows outbound port 25 before you try to test live delivery. Submission on 587 can work fine even while outbound delivery on 25 is blocked at the provider level — that's why SignalMTA's go-live checks keep inbound submission readiness and outbound delivery readiness separate.

2. Listener configuration example

```
{
  "smtp_receive": {
```

```
    "listeners": [
      {
        "id": "submission-587",
        "listen": "0.0.0.0:587",
        "tls": { "mode": "starttls-required", "certificateRef": "file:///etc/signalmta/tls/submission.pem
        "authRequired": true,
        "allowedRelayCidrs": ["10.40.0.0/16", "203.0.113.0/28"],
        "maxMessageBytes": 52428800,
        "rateLimits": { "perAuthUserPerMinute": 20000, "perSourceIpPerMinute": 5000 }
      },
      {
        "id": "lb-private-2525",
        "listen": "10.40.12.10:2525",
        "tls": { "mode": "opportunistic" },
        "authRequired": true,
        "allowedRelayCidrs": ["10.40.12.0/24"],
        "privateOnly": true
      }
    ]
  }
}
```

3. Configure submission in the WebUI

Go to Security -> Relay Control and confirm the default is deny-by-default before you do anything else.

Only create allowed relay CIDRs for application servers, private load balancers, or customer networks that genuinely need to submit mail.

Go to Security -> SMTP AUTH and set up a separate user per application, tenant, or stream. Never share one credential across unrelated customers.

For each SMTP AUTH user, bind the allowed tenants, message streams, source networks, maximum message size, permitted X-Signal controls, and route/IP pool permissions.

Go to Configuration -> Listeners and turn on the submission ports you need. On any public or customer-facing listener, require STARTTLS before AUTH.

Go to Configuration -> Submission Flow and map each authenticated identity to a queue lane, priority class, sending domain, route, and source-IP pool.

Click Validate and clear every warning — open relay, missing TLS, missing AUTH, unsafe CIDR, or route-permission issues — before you publish.

Send one SMTP test message and one API test message, then confirm both show up in Queues, Logs, and Reports with the tenant, stream, route, and source-IP you expected.

4. SMTP submission examples

Reach for SMTP when your customer applications are already wired up to send through an MTA-style endpoint. SignalMTA handles standard RFC-style SMTP conversations, stores the original message

with safe metadata, and applies routing policy before it admits the message to the queue.

```
# Authenticated STARTTLS submission on 587
```

*swaks --server mta.example.com --port 587 --tls *

```
  --auth LOGIN --auth-user acme-transactional --auth-password "$SMTP_PASS" \
  --from bounce+msg_01J123@bounces.example.com \
  --to user@example.net \
  --header "From: Acme Alerts <alerts@example.com>" \
  --header "Subject: Your security code" \
  --header "X-Signal-Stream: transactional" \
  --header "X-Signal-Priority: high" \
  --header "X-Customer-Trace: acct-1934" \
  --body "Your code is 123456"

# Marketing stream with RFC-compliant unsubscribe headers
```

*swaks --server mta.example.com --port 587 --tls *

```
  --auth LOGIN --auth-user acme-marketing --auth-password "$SMTP_PASS" \
  --from bounce+msg_01J124@bounces.example.com \
  --to subscriber@example.net \
  --header "From: Acme News <news@example.com>" \
  --header "Subject: July product update" \
  --header "X-Signal-Stream: marketing-us" \
  --header "X-Signal-Priority: bulk" \
  --header "X-Signal-Pool: warmup-us-a" \
  --header "List-Unsubscribe: <mailto:unsubscribe@example.com>, <https://example.com/u/abc123>" \
  --header "List-Unsubscribe-Post: List-Unsubscribe=One-Click" \
  --body "News content..."
```

5. API injection examples

Reach for API injection when an application needs to send structured metadata, wants idempotency, or needs deterministic route selection without encoding every control into an SMTP header. Under the hood, API injection and SMTP submission share the same admission policy and queueing path.

```
curl -X POST https://mta.example.com/api/mta/enqueue \
  -H "Authorization: Bearer $SIGNALMTA_TOKEN" \
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: order-98271-receipt-v1" \
  -d '{
    "tenant": "acme",
    "stream": "transactional",
    "priority": "high",
    "mailFrom": "bounce+msg_01J125@bounces.example.com",
    "recipients": ["user@example.net"],
    "headers": {
      "From": "Acme Receipts <receipts@example.com>",
      "Subject": "Your receipt",
      "X-Customer-Trace": "order-98271"
    },
    "routing": {
      "route": "direct-us-east",
```

```
    "pool": "pool-acme-dedicated"
  },
  "mime": "From: Acme Receipts <receipts@example.com>\\r\\nTo: user@example.net\\r\\nSubject: Your rece
}'
```

When a caller retries with the same idempotency key, SignalMTA hands back the original accepted message identity instead of queueing a duplicate. If that same key shows up with different content, SignalMTA rejects the request outright as an idempotency conflict.

6. Submission controls and headers

X-Signal-Stream — Select tenant/customer stream. — Controls permissions, routing, reporting, limits, and SignalAI scope.

X-Signal-Priority — Request transactional, standard, bulk, or background handling. — Maps to scheduler reserve and queue-age targets.

X-Signal-Pool — Request an approved source-IP pool. — Rejected if the authenticated sender lacks permission.

X-Signal-Route — Request a named route or delivery-service connector. — Logged on every attempt and bounded by route policy.

Custom X-headers — Carry customer tracking values. — Preserved and exported when policy allows the field.

List-Unsubscribe — Required for compliant bulk streams. — Validated before scale-up and included in deliverability warnings.

SignalMTA reserves the X-Signal-* namespace for its own operational controls. For customer tracking, use customer-owned names instead — things like X-Customer-Trace, X-Campaign-ID, or X-Tenant-Message-ID. Don't let arbitrary senders set route, pool, source-IP, or priority headers unless your policy explicitly grants that identity permission.

7. List-Unsubscribe and one-click unsubscribe

SignalMTA treats unsubscribe headers as compliance-critical metadata, not cosmetic ones. It keeps valid customer-supplied List-Unsubscribe headers intact, can add them itself through an approved stream/list policy, rejects malformed values before letting the message into the queue, and logs one-click metadata for reporting and suppression workflows. It won't invent unsubscribe endpoints for a stream with no configured list or consent policy — you have to define the URL/mailbox that will actually handle the request.

List-Unsubscribe — Comma-separated angle-bracketed mailto: and/or https: URLs, for example <mailto:unsubscribe@example.com>, <https://example.com/u/abc>. — Validated before queue admission. Invalid values are rejected with an invalid-header response instead of being silently sent.

List-Unsubscribe-Post — List-Unsubscribe=One-Click — Marks the message as one-click capable and lets SignalMTA classify callbacks as RFC 8058 one-click unsubscribe events.

Generated policy headers — Configured per tenant, stream, list, or campaign. — SignalMTA may add the headers only when the stream policy contains a real unsubscribe endpoint and the message class allows it.

Transactional/security mail — Usually omitted unless the sender's policy explicitly requires it. — Keep OTP, password reset, receipts, and security notices on separate routes so marketing compliance logic does not alter critical transactional flows.

```
{
  "submissionFlow": {
    "rules": [{
      "name": "marketing requires unsubscribe",
      "match": { "streamType": "marketing" },
      "requireListUnsubscribe": true,
      "set": {
        "listUnsubscribe": {
          "mailto": "unsubscribe@example.com",
          "httpsTemplate": "https://example.com/unsubscribe/{tenant}/{stream}/{recipient_token}",
          "oneClick": true
        }
      }
    }]
  }
}
```

As a rule of thumb: require unsubscribe headers on marketing, lifecycle, newsletter, and promotional streams; keep valid customer-supplied headers intact when the sender owns the list workflow; only generate headers from a configured stream/list policy; and keep complaint, unsubscribe, and bounce suppressions in separate reporting buckets.

8. Common submission rejections

530 authentication required — The listener requires SMTP AUTH or mTLS before relay. — Use port 587 with STARTTLS and valid credentials, or bind the mTLS client certificate to a stream.

538 encryption required — The listener does not allow AUTH before STARTTLS. — Confirm the client sends STARTTLS before AUTH and that the certificate chain is valid.

550 relay denied — The source CIDR, identity, tenant, stream, or route is not permitted. — Review Security -> Relay Control and the authenticated user's stream/route permissions.

552 message too large — The message exceeds listener, tenant, or stream size limits. — Check max message bytes and attachment policy.

554 policy rejection — SSL, suppression, required headers, malformed unsubscribe, or submission-flow policy rejected the message. — Open Logs -> Admission and inspect the policy trace and rejection reason.

421 admission temporarily limited — Queue admission, per-identity rate, spool watermark, or safety gate is temporarily limiting new submissions. — Inspect queue age, spool watermark, admission limits, and source identity rate caps.

9. What operators should verify before live traffic

Make sure every exposed listener has a clear purpose, TLS policy, relay rule, rate limit, and owner assigned to it.

No customer-facing listener should accept unauthenticated relay from the public internet.

SMTP AUTH users and API tokens need to be scoped down to specific tenants, streams, and routes.

Marketing streams need a valid unsubscribe policy in place before you ramp up volume.

Submission logs should capture tenant, stream, auth identity, source IP, listener id, idempotency key where it applies, and the queue result.

You should be able to trace one SMTP test message and one API test message all the way from admission through queue placement, route selection, delivery attempt, webhook export, and report aggregation.

Sending Domains, DKIM, DKIM2, TLS, and DNS

A sending domain isn't just a From address — think of it as an identity bundle: visible From domain, return-path domain, DKIM selectors, SPF/DMARC alignment, how your TLS is set up, VERP correlation, DNS verification, allowed streams, and signing evidence. Don't ramp up volume on a stream until that whole bundle is complete and you've verified it from outside your own network.

SignalMTA configuration WebUI view

This is the actual WebUI configuration screen you'd use to check domain, routing, authentication, queue, and validation settings.

1. Create the domain plan

Begin with a domain-auth plan. It generates the DNS records, selector material, return-path target, verification checks, and stream binding for you. If you're a service provider, add a platform selector so SignalMTA can double-sign messages with both the customer identity and the platform identity.

```
./bin/signalmta domain-auth-plan \
  --domain mail.customer.example \
  --from-domain customer.example \
  --selector smta1 \
  --second-domain signalmta.example \
  --second-selector sig2026 \
  --bounce-domain b.customer.example \
  --dmarc-policy quarantine \
  --tls-policy opportunistic
```

WebUI setup: create and verify DKIM without using the CLI

You don't need the CLI to set up DKIM — the WebUI can do the whole thing. It runs on the same domain-auth planner and validation engine as the CLI, just wrapped in a guided workflow with copy-ready DNS records, verification checks, and publish controls.

Head to Domains and hit Add sending domain to get started.

Fill in the visible From domain, the return-path/bounce domain, the allowed tenant or stream, and the default IP pool or route.

Under DKIM signing, decide whether SignalMTA should generate a new selector or attach an existing one. For a brand-new domain, Generate selector is usually the right call.

Pick the selector name, algorithm, and key storage method. Go with stable selector names like smta1 or sig2026, and store private keys as secret references — never pasted plaintext.

If you're running as a service provider and need two signatures, turn on Double DKIM signing and pick both the customer-domain selector and the platform-domain selector.

Click Generate DNS records, and the WebUI will show you the DKIM TXT record, SPF guidance, DMARC guidance, return-path MX/CNAME records, and optional TLS reporting records.

Publish those DNS records with the domain's DNS provider as-is. Don't modify the DKIM public key, wrap it manually, or change the selector hostname.

Back in the WebUI, click Verify DNS and keep running it until DKIM selector visibility, key parsing, return-path reachability, SPF syntax, DMARC syntax, and alignment checks all pass.

Click Bind to streams and choose which streams, tenants, routes, pools, or virtual instances are allowed to use this domain identity.

Click Validate, look over the diff and reload impact, then click Publish. SignalMTA logs the actor, reason, revision, and DKIM selector binding in the audit trail.

Selector — smta1, smta2, or a date/versioned selector. — Stable selectors make rotation and troubleshooting easier.

Algorithm — RSA SHA-256 for broad compatibility; Ed25519 where receiver compatibility is acceptable. — RSA remains the safest default for heterogeneous mailbox-provider delivery.

Private key storage — Secret reference such as env://, file://, Vault, AWS, Azure, GCP, or Kubernetes Secret. — Prevents private keys from appearing in config diffs, screenshots, exports, or support bundles.

Signed headers — From, To, Subject, Date, Message-ID, MIME-Version, Content-Type, List-Unsubscribe where applicable. — Creates consistent authentication evidence while avoiding unstable headers.

Stream binding — Bind only the streams that should use the identity. — Prevents a marketing stream from accidentally using a transactional identity.

Double signing — Enable only when both customer and platform identity are required. — Useful for service providers, but it must be visible in attempt logs and selector validation.

Once it's published, go to Domains -> Domain detail -> Authentication evidence to check DKIM pass/fail state, which selector was used, the signing domain, the aligned From domain, double-signing state, and recent delivery-attempt examples. If a selector's failing, the WebUI tells you the exact hostname it checked, the resolver result, the parsed key state, and a recommended fix.

2. Publish the DNS records

Publish exactly what the plan gives you — no edits. Keep DKIM selectors short and stable. Give VERP its own dedicated return-path domain so async bounces can be tied back to the original message, stream, source IP, route, and attempt. Point DMARC reports at addresses your team actually watches.

Customer DKIM — smta1._domainkey.customer.example TXT "v=DKIM1; k=rsa; p=..." — Signs with the customer-visible identity.

Platform DKIM — sig2026._domainkey.signalmta.example TXT "v=DKIM1; k=ed25519; p=..." — Optional second signature for provider/platform identity.

SPF — customer.example TXT "v=spf1 ip4:203.0.113.44 include:spf.customer.example -all" — Authorizes the visible infrastructure where appropriate.

DMARC — _dmarc.customer.example TXT "v=DMARC1; p=quarantine; rua=mailto:dmarc@customer.example" — Defines alignment policy and aggregate reporting.

Return-path MX — b.customer.example MX 10 inbound-bounce.customer-host.example — Receives DSNs and out-of-band bounces.

MTA-STS/TLS reporting — _smtp._tls.customer.example TXT "v=TLSRPTv1; rua=mailto:tlsrpt@customer.example" — Supports TLS configuration reporting when used by the domain.

3. Bind the domain to streams and routes

Binding is what decides which traffic can actually use this identity. You don't want a transactional stream accidentally inheriting a marketing return path, and you don't want a warmup stream borrowing a production-critical selector unless you meant to do that.

```
{
  "authentication": {
    "sendingDomains": [{
      "domain": "customer.example",
      "tenant": "acme",
      "allowedStreams": ["transactional", "marketing-us"],
      "returnPathDomain": "b.customer.example",
      "dkimSelectors": ["smta1"],
      "doubleSignWith": { "domain": "signalmta.example", "selector": "sig2026" },
      "defaultPool": "pool-east-a",
      "requireAlignedFrom": true
    }]
  }
}
```

4. Verify before sending

Check from more than one resolver — a local DNS cache can mask propagation problems you'd otherwise catch. Before you're cleared to scale up, the verification command needs to pass DKIM visibility, return-path MX, SPF syntax, DMARC syntax, alignment rules, selector key parsing, and bounce-domain reachability.

```
./bin/signalmta domain verify customer.example --resolvers 1.1.1.1,8.8.8.8,9.9.9.9
./bin/signalmta domain evidence customer.example --format json
./bin/signalmta logs tail --type delivery-attempt --domain gmail.com --fields dkim,tls,auth,route
```

5. Configure how TLS is set up

By default, outbound internet delivery uses opportunistic TLS. Switch to required TLS for routes covered by contract or for known partner domains. In required mode, SignalMTA needs to defer the message rather than quietly downgrade if STARTTLS, SNI, certificate validation, MTA-STS, or DANE policy fails.

Opportunistic — General mailbox-provider delivery. — Attempt STARTTLS, log negotiated version/cipher, continue if unsupported.

Required — Partner route, regulated stream, internal relay, security-sensitive tenant. — Defer if TLS cannot be negotiated or verified.

Disabled — Only in a controlled lab or smart-sink target. — Should fail production safety checks for public sending.

```
"tlsPolicy": {
  "minVersion": "TLSv1.2",
  "rejectUnauthorized": true,
  "requireTlsForAuth": true,
  "mtaSts": { "enabled": true, "enforceMode": "testing" },
  "dane": { "enabled": false, "requireDnssecValidation": true }
}
```

6. Rotate selectors safely

Selector rotation follows five steps: publish, verify, cut over, observe, retire. Publish the new selector first, wait for public resolvers to pick it up, switch the stream bindings over, watch delivery-attempt logs for both old and new selectors during the changeover, and only retire the old one once every queued message referencing it has drained.

```
./bin/signalmta dkim selector create --domain customer.example --selector smta2 --algorithm rsa-sha256
./bin/signalmta dkim selector verify --domain customer.example --selector smta2
./bin/signalmta dkim selector activate --domain customer.example --selector smta2 --previous smta1 --grac
```

Common authentication problems

DKIM record not found — Selector typo, DNS not propagated, wrong zone. — Compare the plan output with public resolver lookups and verify CNAME flattening is not altering TXT records.

DMARC alignment fail — From domain and return-path/signing identity do not align as expected. — Use aligned DKIM domain, aligned return path, or adjust stream identity policy.

TLS required deferrals — Remote MX lacks STARTTLS or certificate validation fails. — Use opportunistic mode for that destination or fix the partner TLS policy before retrying.

Double signing missing — Second selector not bound to the stream or secret reference missing. — Check selector binding, secret provider, and delivery-attempt signing evidence.

Queueing and Spool Integrity

Everything you accept lives or dies in the queue, so that's where reliability actually comes from. SignalMTA writes each accepted message to durable storage before it attempts delivery, then decides what to send next based on priority, recipient-domain/MX, stream policy, source-IP pool, retry state, and any operator holds.

Don't let RAM, an external message broker, or a reporting database stand in as the authoritative store for accepted mail. RAM is fine for counters, scheduler candidate windows, DNS cache, and live telemetry, since you can always rebuild those. But the accepted message body, queue metadata, route decision, retry state, bounce context, and delivery-attempt ledger have to live in the SignalMTA spool. Otherwise a restart, worker crash, broker disconnect, or analytics outage can quietly lose mail.

Ready — Eligible for delivery. — Inspect workers, shaping, and domain capacity if age rises.

Deferred — Waiting for retry after remote or local condition. — Review category, retry time, and affected scope.

Held — Paused by operator, policy, warmup, or SignalAI advisory. — Release only after confirming the reason is resolved.

Permanent failure — No more automatic retries. — Redrive only after root-cause correction and recipient eligibility checks.

Quarantine — Integrity check failed. — Run spool inspection and preserve evidence.

```
./bin/signalmta queue summary --group-by priority,domain,stream
./bin/signalmta queue inspect --domain gmail.com --state deferred --limit 100
./bin/signalmta queue hold --stream promotional --reason "provider deferrals"
./bin/signalmta queue release --stream promotional --max 10000
./bin/signalmta spool-maintenance --check --repair-safe
```

Priority reserve

Set aside capacity for high-priority streams so bulk traffic can't eat every worker slot, DNS lookup, source-IP allowance, or bit of spool bandwidth. Watch queue age as your main health signal: if a high-priority queue's age keeps climbing, that needs attention even when overall throughput looks fine.

Scheduler candidate windows

Once queue depth gets large, you don't want a delivery cycle scanning every queued metadata file just to pick its next batch. SignalMTA works with bounded candidate windows instead: each drain cycle looks at a configured slice of queued metadata, rotates which shard gets scanned first, scores the candidates by priority, queue lane, and age, then applies fairness caps before handing work to delivery workers. That protects a path for high-priority mail while stopping any single bulk stream, destination domain, pool, or static source IP from hogging the host.

```
"realMta": {
  "maxDeliveriesPerTick": 1000,
  "maxDeliveryConcurrency": 400,
  "queueScheduler": {
```

```
    "candidateWindowMultiplier": 20,
    "maxCandidates": 50000,
    "maxPerDomainPerTick": 0,
    "maxPerPoolPerTick": 0,
    "maxPerSourceIpPerTick": 0,
    "highPriorityReservePercent": 35
  }
}
```

candidateWindowMultiplier — Controls how many queued messages are considered relative to maxDeliveriesPerTick. — Increase when priority traffic is spread across many shards; reduce when metadata read pressure is high.

maxCandidates — Hard cap on metadata considered per drain cycle. — Keep high enough to find priority mail, low enough to avoid large queue scans during backlog.

maxPerDomainPerTick — Caps one recipient domain or provider family inside one drain cycle. — Use explicit caps for major providers when one domain can dominate traffic.

maxPerPoolPerTick — Caps one IP pool inside one drain cycle. — Use when shared pools and dedicated pools must both make progress.

maxPerSourceIpPerTick — Caps one source IP inside one drain cycle. — Use to prevent a single static IP or proxy egress from absorbing too much work.

highPriorityReservePercent — Reserves part of each drain cycle for high-priority or transactional lanes. — Start around 25-35%; increase if transactional queue age rises during bulk sends.

AMQP and broker-based injection

SignalMTA can pull from AMQP-compatible brokers like RabbitMQ for injection workflows and event integration. That's handy when a customer's application already publishes campaign work into queues, when submitters need asynchronous backpressure, or when a downstream system wants delivery, bounce, complaint, unsubscribe, and suppression events delivered into a broker it already runs.

AMQP is not the authoritative MTA queue once a message is accepted. The safe pattern: consume from AMQP, validate the submission, write the accepted message and its metadata to the SignalMTA spool, and only then acknowledge the broker message. If the durable spool write fails, SignalMTA should NACK or requeue that broker message rather than acknowledge it, so a broker ack never turns into a false delivery guarantee.

Set up the exchange, queue, routing key, tenant, stream, and default submission route.

Make the AMQP payload carry sender identity, recipients, a MIME or template reference, an idempotency key, and any optional X-Signal-* controls.

Check message size, header safety, sending-domain permission, suppression policy, List-Unsubscribe requirements, and route permissions.

Commit the message body and its metadata into the SignalMTA spool.

Only acknowledge AMQP once that spool commit has succeeded.

Push the resulting delivery attempts, bounces, complaints, and suppressions out through webhooks, JSONL, Kafka, AMQP, or whichever adapter the customer picked.

Major-domain queue handling

Big mailbox providers need scheduling that's aware of who they are. Group domains and MX patterns into provider families, then shape traffic by family, recipient domain, MX rollup, stream, pool, source IP, and priority. For instance, Gmail-bound traffic sitting on a warmup pool shouldn't be allowed to soak up all the worker slots that Microsoft-bound transactional mail needs, and throttling one source IP shouldn't slow down an unrelated source IP in the same pool unless your policy says the whole pool is affected.

Queue age rising for one provider family — Provider-specific capacity, throttling, block, or warmup limit. — Reduce only the affected recipient-domain/MX/pool/source-IP scope and keep other traffic moving.

High-priority age rising while bulk flows — Priority reserve or lane isolation is too low. — Increase high-priority reserve, reduce bulk candidate share, and review stream caps.

One pool dominates selected candidates — Pool fairness cap is too high or route distribution is unbalanced. — Set maxPerPoolPerTick and review submission-flow route rules.

One source IP dominates attempts — Dedicated-IP or proxy egress path is absorbing shared work. — Set maxPerSourceIpPerTick and review pool membership.

DNS latency rises with backlog — Resolver pressure is limiting delivery speed. — Use local recursive resolvers, tune cache TTLs, and reduce unnecessary unique-domain pressure.

Traffic Shaping, Throttling, Retry, and Warmup

Traffic shaping is what lets SignalMTA keep throughput, sender reputation, and host stability intact all at once. Shape at the smallest scope that's actually affected: recipient-domain/MX before global, source IP before pool, stream before tenant, and error category before a blanket retry rule.

SignalMTA policy WebUI view

This is the actual WebUI policy screen for runtime controls, traffic shaping, retry behavior, and operator guardrails.

1. Start with conservative baseline limits

For every domain or provider group, start with caps well under the host's ceiling. Predictable behavior matters more than squeezing out maximum throughput at the start. Keep separate defaults for major mailbox providers, regional ISPs, corporate domains, and anything unknown.

```
{
  "traffic_shaping": {
    "defaults": {
      "max_connections": 8,
      "max_messages_per_minute": 2500,
      "messages_per_connection": 50,
      "retry_jitter_percent": 17
    },
    "domains": {
      "gmail.com": {
        "max_connections": 80,
        "max_messages_per_minute": 45000,
        "messages_per_connection": 75,
        "max_ready_queue_age_seconds": 180
      },
      "orange.fr": {
        "max_connections": 14,
        "max_messages_per_minute": 6000,
        "regional_profile": "FR",
        "localized_diagnostics": true
      }
    }
  }
}
```

2. Shape by domain, MX, pool, source IP, stream, and priority

How a mailbox provider behaves is almost always tied to the recipient domain/MX, not the whole world. A Gmail queue on one pool shouldn't automatically get punished because of unrelated traffic on another pool, unless the evidence points to a genuine global stream problem. And for high-priority messages, hold back worker and DNS capacity so bulk mail can't starve the urgent stuff.

Domain/MX — Connections, messages/minute, retry curve, cooldown, TLS mode. — Provider limits, greylisting, rate-limit replies, MX-specific behavior.

Pool — Total concurrent connections, max messages/minute, overflow behavior. — Separate warmup, production, transactional, and cooldown pools.

Source IP — Warmup cap, pause, cooldown, max connections, reputation state. — Protect an individual IP without slowing the whole pool.

Stream — Submission cap, priority, route permissions, suppression policy. — Keep one customer/campaign from consuming shared capacity.

Priority — Reserved scheduler share, max age target, escalation threshold. — Guarantee capacity for transactional or high-value traffic.

3. Turn on adaptive controls with explicit guardrails

An Adaptive Delivery Policy is the operator-approved rule set that tells SignalMTA exactly when it's allowed to adjust delivery behavior on its own. It's not some vague “AI mode” or a hidden global throttle sitting behind the scenes. It's a scoped policy with explicit evidence triggers, an affected scope, an allowed action, a cooldown length, recovery behavior, and an audit trail.

Adaptive shaping needs to react to evidence, not hunches. Spell out the trigger, the scope it touches, the reduction it applies, how long the cooldown lasts, the recovery step, and the ceiling on what it's allowed to automate. SignalAI can suggest changes, but guarded auto-actions should only fire inside the policy you've approved. Start in advisory-only mode, check its recommendations against delivery-attempt logs, and only let it act automatically for narrow, low-risk cases, like temporarily throttling one domain/MX/pool grouping.

Trigger — Enhanced status code, bounce category, deferral rate, queue age, complaint rate, TLS/auth failure, proxy mismatch, or warmup threshold. — Use evidence that appears in logs and reports so the action can be explained later.

Scope — Domain, MX rollup, source IP, pool, stream, tenant, route, connector, provider grouping, or error class. — Choose the smallest scope that explains the problem. Avoid global changes unless the whole installation is affected.

Action — Reduce rate, lower connections, change retry schedule, freeze warmup, pause source IP, hold stream, or require human approval. — Prefer reversible actions first. Holds should include a reason and review owner.

Cooldown — How long the action remains in force before reevaluation. — Use enough time for provider behavior to stabilize. Do not oscillate rates every few seconds.

Recovery — Step-up percentage, reevaluation interval, maximum recovery rate, and conditions that must be healthy. — Recover gradually when acceptance improves, not all at once.

Authority — Advisory-only, guarded automatic, or human-approval-required. — Use advisory-only for new policies and human approval for reputation, complaint, or block-oriented actions.

```
{
  "adaptiveThrottling": {
    "enabled": true,
    "throttleFloorPercent": 35,
    "throttleRecoveryPercent": 12,
    "triggers": [
```

```
      {
        "match": { "enhancedStatusPrefix": "4.7", "category": "remote-throttling" },
        "scope": "domain_mx_pool",
        "reducePercent": 35,
        "cooldown": "30m",
        "recoverEvery": "10m",
        "maxAutoReductions": 3
      },
      {
        "match": { "category": "block_oriented" },
        "scope": "stream_domain_pool",
        "action": "hold_for_review"
      }
    ]
  }
}
```

4. Use error-class-aware retry schedules

Your retry schedule needs to fit the failure it's responding to. Greylisting deserves a quick retry. Provider rate limits call for slower retries paired with throttling. Block-oriented and spam-oriented failures should hold or pause that stream until a deliverability operator looks at the evidence. And recipient-invalid errors shouldn't be retried at all.

Greylisting — 10m, 30m, 90m, 4h, then normal backoff. — Do not permanently suppress recipients.

Remote throttling — 15m, 45m, 2h, 6h, 12h with throttle reduction. — Do not increase concurrency while deferrals rise.

Bad recipient — No retry; suppress according to stream policy. — Do not retry 5.1.x mailbox failures.

Spam/block — Hold affected stream/domain/pool for review. — Do not route around a block without understanding the cause.

TLS/auth policy — Defer until configuration is corrected. — Do not downgrade required TLS or ignore DKIM/DMARC failures.

```
hook smtp_reply(reply, message, domain, stream) {
  if reply.status ~= "4.7." {
    throttle scope: domain.mx reduce: 35% for 30m;
    retry schedule: "15m,45m,2h,6h,12h";
    retry jitter: 22%;
    emit "throttle.remote_rate_limit";
  }
  if reply.status ~= "5.1." {
    suppress recipient: message.recipient;
    stop retry;
    emit "suppression.bad_recipient";
  }
  if reply.category == "block_oriented" or reply.category == "spam_oriented" {
    hold stream: stream.id reason "deliverability review required";
    notify "deliverability";
  }
}
```

5. Automate new-IP warmup without hiding risk

SignalWarmup runs automated by default. An operator sets up the warmup plan once: source IP, pool, start date, initial cap, target cap, recipient groupings, overflow pool, and guardrails. From there, SignalWarmup keeps checking delivery evidence, raises caps when things look healthy, freezes or pauses only the affected scope when risk shows up, and logs every decision it makes. Operators can still override the controller, but they shouldn't need to babysit a warmup spreadsheet every day.

```
{
  "ipWarmup": {
    "enabled": true,
    "controller": "SignalWarmup",
    "mode": "automatic-with-operator-override",
    "defaultAlgorithm": "token-bucket",
    "policyDistribution": "websocket-and-config-revision",
    "requireFutureActivation": true,
    "requireWarmOverflowPool": true,
    "guardrails": {
      "maxDailyIncreasePercent": 70,
      "freezeOnComplaintRatePercent": 0.08,
      "freezeOnHardBounceRatePercent": 4.5,
      "freezeOnDeferralRatePercent": 18,
      "freezeOnBlockSignals": true,
      "requireReverseDns": true,
      "requireDkimSpfAlignment": true,
      "requireTlsHealthy": true
    },
    "plans": [{
      "ip": "203.0.113.44",
      "pool": "pool-new-dedicated",
      "proxy": "egress-east-a",
      "mode": "AUTO",
      "startDate": "2026-07-20",
      "initialDailyCap": 1000,
      "targetDailyCap": 250000,
      "overflowPool": "pool-established",
      "groupings": ["gmail-primary", "microsoft-consumer", "yahoo-aol"]
    }]
  }
}
```

The delivery scheduler is what actually enforces warmup in the mail path. Once the current cap for a source IP and recipient grouping is used up, any additional eligible messages shift to the configured warm overflow pool or get held under policy. That keeps time-sensitive mail moving while still protecting the new IP's reputation. For the big providers, warmup is grouped by provider family rather than by raw recipient domain alone, since Gmail-hosted domains, Microsoft consumer/business domains, Yahoo/AOL, Apple, Comcast, and regional ISPs can each behave quite differently. SignalMTA's warmup reporting shows the daily cap, delivered volume, cap utilization, bounce rate, complaint rate, 4.7.x deferral rate, block-oriented replies, authentication state, the next cap decision, and whether the controller plans to increase, wait, freeze, or pause.

Complaint rate — Below configured stream/provider threshold. — Increase the cap only when complaint trend remains stable; otherwise freeze the affected grouping.

Hard-bounce rate — Low and explained by normal list churn. — Pause the stream or recipient cohort when list quality is suspect.

Remote throttling — 4.7.x replies stay below the provider threshold. — Hold the provider-group cap, reduce source-IP rate, and extend retry jitter automatically.

Block-oriented replies — None during normal warmup. — Pause the affected source IP/provider group and require deliverability review.

Authentication/rDNS — DKIM/SPF/DMARC/rDNS pass before scale-up. — Block live warmup growth until fixed; continue routing eligible overflow to a warm pool.

6. Verify shaping is working

Pull up the domain drilldown, queue age, retry plan, and delivery-attempt logs together. A shaping change that's actually helping will lower deferrals for the affected scope without pushing up queue age on high-priority streams. Whatever you export should show the same category, next retry, throttle id, source IP, pool, and SignalAI recommendation that shows up in the WebUI.

```
./bin/signalmta shaping explain --domain gmail.com --pool pool-east-a
./bin/signalmta queue summary --group-by priority,domain,pool,state
./bin/signalmta logs tail --type delivery-attempt --domain gmail.com --category remote-throttling
GET /api/mta/domains/gmail.com/shaping/evidence
```

IP Pools, Dedicated IPs, and Service-Provider Routing

IP strategy is one of the biggest operating decisions you'll make running a high-volume MTA. SignalMTA lets you model shared pools, dedicated customer IPs, warmup pools, cooldown pools, provider-specific pools, and proxy-owned egress IPs, all while keeping reporting and delivery evidence tied back to the right tenant, stream, route, source IP, and provider response.

1. Choose the right IP model

Shared pool — Multiple customers or streams can safely share reputation and volume. — Group similar traffic quality together. Do not mix transactional mail with risky acquisition campaigns.

Dedicated IP — A customer, business unit, or stream needs isolated reputation and reporting. — Bind the tenant/stream to one or more source IPs and keep bounce/complaint reporting separated.

Warmup pool — New IPs need gradual volume growth before joining production pools. — Use daily caps, provider grouping, complaint thresholds, hard-bounce thresholds, and automatic pause rules.

Cooldown pool — An IP or pool shows elevated deferrals, blocks, or complaint risk. — Route only controlled recovery traffic and require operator review before returning to production.

Provider-specific pool — Major destinations need separate treatment. — Use when Gmail, Microsoft, Yahoo, regional ISPs, or B2B domains behave differently enough to justify separate caps.

External relay/provider connector — Some mail should go through SendGrid, Mailgun, SparkPost, or another delivery provider. — Tag the route, preserve customer metadata, and ingest provider bounces/complaints back into SignalMTA reporting.

2. Configure shared pools and dedicated IPs

Reach for shared pools when traffic carries similar risk, permission, and expected engagement. Reach for dedicated IPs when a customer has bought or needs isolated sending reputation. A dedicated IP can still sit inside a pool, but route policy needs to stop unrelated tenants from touching it.

```
{
  "ipPools": [
    {
      "id": "pool-shared-transactional-us",
      "type": "shared",
      "sourceIps": ["203.0.113.10", "203.0.113.11"],
      "allowedStreams": ["transactional"],
      "maxConnections": 400,
      "maxMessagesPerMinute": 120000
    },
    {
      "id": "pool-acme-dedicated",
      "type": "dedicated",
      "tenant": "acme",
      "sourceIps": ["203.0.113.44", "203.0.113.45"],
      "allowedStreams": ["acme-transactional", "acme-marketing"],
      "warmupPlan": "new-ip-standard",
```

```
      "overflowPool": "pool-acme-established"
    }
  ]
}
```

3. Route message streams to the correct IPs

Decide the IP pool during submission, before the message is even admitted to the queue. That decision then rides along as message metadata, shows up in delivery-attempt logs, and feeds reporting. Service providers should base route decisions on tenant, stream, customer tier, requested X-Signal headers, API token, sending domain, campaign tag, and compliance state.

```
{
  "submissionFlow": {
    "rules": [
      {
        "name": "Acme dedicated transactional",
        "match": { "tenant": "acme", "stream": "transactional" },
        "set": {
          "priority": "high",
          "ipPool": "pool-acme-dedicated",
          "sendingDomain": "mail.acme.example",
          "tlsMode": "required"
        }
      },
      {
        "name": "Shared marketing default",
        "match": { "streamType": "marketing", "tenantTier": "standard" },
        "set": {
          "priority": "bulk",
          "ipPool": "pool-shared-marketing-us",
          "requireListUnsubscribe": true
        }
      }
    ]
  }
}
```

4. Use X-Signal headers safely

Customers can ask for routing hints, but authenticated route policy always has the final say. For example, a customer might submit X-Signal-Pool: pool-acme-dedicated, but SignalMTA will reject or override that if the customer isn't actually permitted to use that pool.

```
X-Signal-Stream: acme-transactional
X-Signal-Priority: high
X-Signal-Pool: pool-acme-dedicated
X-Signal-Route: direct-us-east
X-Customer-Campaign: password-reset
```

5. Manage source IPs outside the MTA host

Some service providers would rather manage source IPs at a proxy or network layer than bind every IP directly to the MTA's operating system. In that setup, SignalMTA picks the egress intent and proxy route, while the proxy owns the actual source IP. Delivery logs still capture both the MTA node and the selected egress identity, so support teams can trace exactly what happened.

```
{
  "outboundProxy": {
    "enabled": true,
    "mode": "haproxy-or-custom",
    "selection": "ssl-route-policy",
    "proxies": [
      { "id": "egress-east-1", "address": "10.20.10.12:2525", "sourceIpPool": "pool-acme-dedicated" },
      { "id": "egress-east-2", "address": "10.20.10.13:2525", "sourceIpPool": "pool-shared-marketing-us"
    ]
  }
}
```

6. IP warmup and reputation protection

Warmup decisions should follow the evidence. Only raise caps when delivery acceptance, hard-bounce rate, complaint rate, authentication, reverse DNS, how TLS is set up, and provider deferrals all look healthy. And pause only the specific grouping that's in trouble whenever you can — one source IP, provider group, domain, or stream shouldn't automatically shut down everything else.

```
{
  "warmup": {
    "plans": [{
      "name": "service-provider-new-dedicated-ip",
      "startDailyCap": 1000,
      "targetDailyCap": 250000,
      "growthPercentPerDay": 25,
      "groupBy": ["sourceIp", "providerGroup", "stream"],
      "pauseOn": {
        "complaintRatePct": 0.08,
        "hardBounceRatePct": 3.0,
        "remoteThrottleRatePct": 12.0,
        "blockCategoryCount": 1,
        "authFailure": true
      }
    }]
  }
}
```

7. What to monitor by IP

Remote acceptance rate by source IP and recipient domain — Shows whether one source IP is receiving weaker SMTP acceptance at a destination. — Reduce rate, move to cooldown, or inspect provider replies.

4.7.x deferrals by IP/pool — Indicates rate pressure or provider throttling. — Apply adaptive throttle to the smallest affected scope.

Block-oriented replies — Reputation or policy issue, often IP or domain specific. — Hold the affected stream/pool and review evidence.

Complaint rate — Recipient-negative signal that can damage pooled reputation. — Pause warmup, suppress recipients, and review stream quality.

Reverse DNS and auth alignment — Provider trust signal and troubleshooting evidence. — Do not scale IPs with missing rDNS or failed DKIM/SPF/DMARC.

8. IP utilization, tenancy, and bounce reporting in the WebUI

The WebUI treats IP usage as something you operate against, not just an inventory list to glance at. An operator should be able to answer who owns or shares a given IP, which streams are allowed to use it, how much of its current capacity is in use, whether it's in warmup, cooldown, or production, and which bounce or complaint categories are tied to it.

SignalMTA routes and IP pool WebUI view

This is the actual WebUI routing screen used to inspect tenant, stream, route, pool, source-IP, and proxy assignments.

Tenancy — Shared, dedicated customer, dedicated stream, leased range, or proxy-managed. — Prevents an unrelated customer from using an isolated reputation lane.

Utilization — Current delivered/minute, connections, queue age, warmup cap usage, and pool capacity. — Shows whether an IP is idle, saturated, warming, cooling, or blocked.

Bounce mix — Bad-recipient, rate-limit, block-oriented, spam-oriented, authentication, TLS, and localized/regional categories. — Separates recipient hygiene problems from reputation or provider-capacity problems.

Complaint/unsubscribe rate — FBL complaint and unsubscribe events tied to source IP, stream, and campaign/tag. — Supports warmup decisions and customer-facing deliverability review.

Route evidence — Tenant, stream, virtual instance, source IP, pool, proxy id, lease id, DKIM selector, TLS mode. — Lets support reconstruct exactly why a message used a specific IP.

The Reports view rolls all of this up by tenant, stream, domain, source-IP pool, and source IP. The Logs view keeps the per-attempt evidence, and exports carry the same join keys so Postmastery-style dashboards or data warehouses can arrive at the same conclusions the WebUI shows.

Bounce Handling, Feedback Loops, and Unsubscribes

SignalMTA takes SMTP replies, DSNs, ARF complaints, and unsubscribe events and turns them into outcomes you can actually read and act on. As an operator you want answers to a handful of questions: what happened, who got hit, should this recipient be suppressed, is it safe to retry, and what evidence backs that call.

SignalMTA reports WebUI view

This is the actual WebUI reports view you'll use for bounce, complaint, unsubscribe, domain, and trend analysis.

1. Understand the two bounce paths

Synchronous failures happen right inside the SMTP transaction, so SignalMTA classifies them on the spot using the SMTP code, enhanced status code, remote MX, provider text, TLS/auth evidence, and retry context. Asynchronous bounces show up later, usually as a DSN email sent back to your return-path domain. That's why you want VERP in place — it's how SignalMTA ties that later DSN back to the original message and its attempt history.

SMTP reply — Code, enhanced status, reply text, MX, route, source IP, TLS, DKIM, attempt number. — Immediate retry, suppress, hold, or throttle decision.

Out-of-band DSN — Return path token, DSN fields, original recipient, action, status, diagnostic code, original message id. — Post-acceptance failure handling and historical reporting.

ARF complaint — Feedback type, source IP, reported domain, original recipient, original envelope id. — Complaint suppression and compliance reporting.

Unsubscribe — List/stream, recipient, method, timestamp, request source. — List-specific suppression and compliance export.

2. Configure VERP return paths

Your return path needs to carry enough information to trace the message, but without leaking sensitive customer data. SignalMTA handles this with tokenized VERP values protected by HMAC and given a TTL. If that token is missing or has expired, correlation falls back to Message-ID, X-Signal trace headers, the original envelope id, DSN recipient fields, and provider queue ids.

```
{
  "bounceProcessor": {
    "outOfBandBounces": true,
    "verp": {
      "enabled": true,
      "mode": "hmac-tokenized",
      "tokenTtlDays": 45,
      "returnPathTemplate": "b+{tenant}.{stream}.{token}@bounces.customer.example",
      "fallbackHeaderCorrelation": true
```

```
    }
  }
}
```

3. Use category-specific actions

Don't lump every 5xx reply together — they mean different things. A 5.1.1 user-unknown is a recipient hygiene problem. A 5.7.1 policy block could point to sender reputation, content, authentication, or something specific to that provider. A 4.7.0 rate-limit reply is almost always a shaping issue, not a bad address. SignalMTA keeps these categories distinct so you know what to do instead of having to guess.

Recipient invalid — 5.1.x, user unknown, mailbox disabled. — Suppress recipient, stop retry. — Confirm this is not a catch-all or temporary directory issue.

Rate limited — 4.7.x, too many connections, try later. — Retry with throttle and jitter. — Inspect recipient-domain/MX, pool, source IP, and queue age.

Block-oriented — Blocked IP/domain, reputation block, deny-list language. — Hold affected scope, alert deliverability. — Check source IP, domain, stream, recent complaints, and block evidence.

Spam-oriented — Content policy, bulk policy, spam-like phrase. — Slow or hold stream, preserve evidence. — Compare content, complaint trend, List-Unsubscribe, and authentication.

Authentication — DKIM/SPF/DMARC/TLS failures, alignment text. — Pause scale-up until fixed. — Run domain-auth verification and inspect signing logs.

Localized/regional — Non-English text, double-byte diagnostics, provider-specific phrases. — Classify by code first, provider phrase map second. — Preserve raw UTF-8 text and review confidence.

4. Process complaints and unsubscribes separately from bounces

A feedback-loop complaint is an abuse or compliance signal, not a delivery failure, so treat it that way. It should generate a complaint event, suppress the recipient whenever the provider tells you who it was, show up in complaint-rate reporting, and get exported to webhooks. A one-click unsubscribe, on the other hand, should only suppress the list or stream it came from, unless your tenant policy calls for broader suppression.

```
{
  "bounceProcessor": {
    "feedbackLoops": {
      "enabled": true,
      "arf": true,
      "parseOriginalMessageHeaders": true,
      "trackingHeaders": true,
      "feedbackIdFormat": "{tenant}:{stream}:{tag}:{message_id}",
      "redactedRecipientPolicy": "record-correlated-complaint-without-recipient-suppression",
      "ingestionEndpoint": "/api/mta/feedback-report",
      "suppressComplainingRecipients": true,
      "exportComplaintEvents": true
    }
  },
  "unsubscribeRequests": {
```

```
    "oneClick": true,
    "suppressRecipientFromStream": true,
    "exportComplianceEvents": true
  }
}
```

5. Set up feedback loop processing

Most mailbox providers that run feedback loops will send Abuse Reporting Format reports to an approved mailbox or endpoint once a recipient hits spam. The details differ by provider, but the pattern is always the same: enroll the sending domain, return-path domain, or source IP; prove you own the domain; designate where FBL reports should land; and route those ARF messages into SignalMTA. Get this wired up before you start sending high volume, so you have complaint data on hand during warmup and while you're evaluating reputation.

Set up or pick a monitored FBL mailbox — something like fbl@bounces.customer.example, or an internal ingestion mailbox your ops team owns.

Enroll every sending domain and source-IP pool that'll carry production traffic with whichever mailbox providers and reputation networks matter for your audience.

Forward inbound ARF messages from the FBL mailbox into the SignalMTA ingestion service, or have your mailbox processor POST the raw message straight to POST /api/mta/feedback-report.

Check that at least one provider report still contains the original message headers. SignalMTA needs those headers to match redacted reports back to the right customer stream.

Confirm the WebUI Reports and Deliverability views break out complaint rate by tenant, stream, domain, source-IP pool, and campaign/tag.

Once a message clears queue admission and gets its internal message id, SignalMTA stamps on its own tracking headers. These are separate from any tracking headers the customer supplied, and policy scripts or downstream relays should leave them alone:

```
X-SignalMTA-Message-ID: msg_01J...
X-SignalMTA-Trace-ID: trace_01J...
```

The header looks like this: Feedback-ID: customer-a:marketing-us:summer-sale:msg_01J...

```
X-Signal-Tenant: customer-a
X-Signal-Stream: marketing-us
```

Never put a raw recipient address inside Feedback-ID — it should describe the operational scope, not expose anyone's personal data. Even if a provider redacts who complained, SignalMTA still logs the complaint, ties it back to the message/tenant/stream wherever the headers allow, and exports the event for reporting and SignalAI analysis. Recipient suppression only happens when you can actually identify the recipient.

6. Ingest an ARF report through the API

The ingestion endpoint will take either structured fields or a full raw ARF email — go with the full raw email when you can, since it keeps the original message headers and provider evidence intact. The parser pulls out the message/feedback-report part, the original message or header section, Feedback-Type, Source-IP, Original-Rcpt-To, Original-Envelope-Id, Reported-Domain, Authentication-Results, Feedback-ID, and any SignalMTA tracking headers.

```
POST /api/mta/feedback-report
Content-Type: application/json

{
  "source": "provider-fbl",
  "rawMime": "Content-Type: multipart/report; report-type=feedback-report; boundary=..."
}
```

Feedback-Type: abuse — Classifies the report as a complaint-class event. — Complaint metric, complaint webhook, suppression if recipient is known.

Original-Rcpt-To — Identifies the complaining recipient when the provider includes it. — Creates recipient suppression for the relevant stream/tenant.

X-SignalMTA-Message-ID — Finds the original queued message and delivery evidence. — Correlates complaint to accepted message and trace.

Feedback-ID — Maps provider-redacted reports to tenant, stream, tag, and message id. — Stream-level complaint reporting without guessing.

Source-IP — Connects complaint pressure to the sending IP or pool. — IP/pool-level reputation review and warmup guardrails.

7. Use the WebUI for complaint operations

If you're not living in the CLI, you can run the entire feedback-loop workflow from the WebUI. Open Deliverability Mode and work across Reports, Bounces, Domains, Logs, and SignalAI together. Reports gives you the complaint trend and who's affected. Bounces shows the classification evidence and what suppression action was taken. Domains tells you if a given provider or domain is under complaint pressure. Logs gives you the raw event chain — ingestion source, message id, trace id, source IP, webhook delivery, all of it. SignalAI tells you whether what you're seeing is an isolated blip or part of a wider reputation problem.

Narrow it down by tenant, stream, domain, source-IP pool, and time window.

Open the complaint category and check whether the event is recipient-identified or provider-redacted.

Review the original message correlation, Feedback-ID, source IP, DKIM selector, route, and the latest delivery attempt.

If the recipient is identified, confirm that suppression actually got created and exported.

If it's redacted instead, look at the stream-level complaint rate and pause or throttle just that affected scope if it's over threshold.

Replay or inspect the complaint webhook if an external suppression system or deliverability platform never got the event.

8. Investigate a bounce or complaint spike

Start in Deliverability Mode, narrowing by tenant, stream, domain, category, and time window.

Figure out whether the spike is synchronous SMTP, an out-of-band DSN, an ARF complaint, or unsubscribes.

Open the category drilldown and read through the plain-English explanation along with the raw evidence.

Compare the affected source IPs, pools, DKIM selectors, how TLS is set up, and any recent config changes.

Take the smallest action that fixes it: suppress the bad recipients, throttle the rate limits, hold the scopes causing blocks, or fix authentication/TLS.

Export the evidence as JSON whenever you need to hand it to a customer, a support engineer, or a deliverability partner.

```
./bin/signalmta bounces classify --sample ./dsn.eml --explain
./bin/signalmta reports bounce-drilldown --stream marketing-us --domain orange.fr --since 2h
./bin/signalmta suppressions explain --recipient user@example.net --tenant acme
GET /api/reports/bounces?stream=marketing-us&category=block_oriented&includeEvidence=true
```

Signal Scripting Language

Signal Scripting Language gives technical operators a way to control how email moves from injection through delivery, without touching the MTA runtime itself. Every script gets validated, versioned, and audited, and runs inside bounds SignalMTA enforces, so policy code you write can't take down or destabilize the workers.

injection — Before queue admission. — Reject, tag, choose stream, require header, set priority.

pre_delivery — Before an SMTP attempt. — Select pool, require TLS, hold, throttle, set connector.

smtp_reply — After remote SMTP response. — Classify, retry, suppress, cooldown, notify.

post_delivery — After final outcome. — Emit webhook, annotate event, update reporting fields.

```
hook injection(message) {
  require_header "List-Unsubscribe" when message.stream == "marketing";
  if message.header("X-Customer-Tier") == "platinum" {
    priority high;
    tag "vip";
  }
}

hook pre_delivery(message, domain) {
  if domain.name == "gmail.com" and message.stream == "warmup" {
    route pool "gmail-warmup";
    throttle messages_per_minute 2500;
  }
  if message.priority == "high" {
    tls require;
  }
}

hook smtp_reply(reply, context) {
  if reply.category == "block_oriented" {
    hold context.stream reason "block signal from provider";
    notify "deliverability";
  }
}
```

Logging, Webhooks, Export, and Grafana

Observability and reporting model

- Live operations in the WebUI — configuration plus the real-time Live SMTP Log.
- Current metrics through the CLI and runtime APIs.
- Comprehensive reporting outside the MTA — BigStats.email (recommended) or your own HTTPS analytics endpoint fed by signed webhooks.

Reporting is an external analytics layer, not a function coupled to the MTA management console. Your telemetry remains yours.

CLI metrics

```
./bin/signalmta metrics
./bin/signalmta metrics --json
./bin/signalmta metrics --json --api-base https://mta.example.com:8443 --token "$SIGNALMTA_API_TOKEN"
```

Use --json for scripting and automation, --api-base to target a specific runtime API, and a bearer token for authenticated access.

Webhook receivers

Signed HTTP POST webhooks remain a first-class telemetry path. BigStats.email is the recommended receiver; customer-owned HTTPS receivers remain fully supported. Outbound/source IP fields stay in the event payload for BigStats and third-party receivers.

The legacy /api/reports endpoint is retained temporarily for compatibility but is no longer the strategic reporting surface. Plan new integrations on webhooks and CLI/API metrics.

SignalMTA's logging is built so you can reconstruct exactly what happened to a message. A support engineer should be able to trace it from submission, through queue admission, route selection, DNS, TLS, DKIM, the SMTP attempt itself, retry or final disposition, bounce handling, webhook export, and report aggregation.

SignalMTA logs WebUI view

This is the actual WebUI logs view for connection logs, delivery attempts, events, and export evidence.

1. Know the log families

Submission/admission — Tenant, stream, source, auth identity, idempotency key, accepted/rejected reason. — Prove whether the MTA accepted the message.

Queue transition — Message id, previous state, next state, reason, retry time, hold actor. — Explain why mail is ready, deferred, held, or permanently failed.

SMTP connection — Connection id, remote MX, source IP, proxy route, SMTP phase, TLS negotiation, latency. — Troubleshoot transport and remote-provider behavior.

Delivery attempt — Message id, attempt, domain, MX, pool, source IP, TLS, DKIM, SMTP reply, category, next action. — Core deliverability evidence and external platform ingestion.

Bounce/complaint/unsubscribe — Category, DSN/ARF fields, suppression result, evidence, original message correlation. — Compliance, suppression, reporting, customer support.

Config/security audit — Actor, revision, diff summary, validation result, login/token/OTP events. — Change control and security review.

2. Use stable join keys

Message ids, trace ids, queue ids, idempotency keys, connection ids, route ids, and webhook event ids give you a way to join data without touching raw recipient addresses. Keep those addresses protected, and where you can, use hashed recipient values in analytics you retain long-term.

```
{
  "event": "delivery_attempt",
  "trace_id": "trc_01J1ABCD",
  "message_id": "msg_01J1EFGH",
  "idempotency_key": "order-889172",
  "tenant": "acme",
  "stream": "transactional",
  "recipient_domain": "gmail.com",
  "mx": "gmail-smtp-in.l.google.com",
  "source_ip": "203.0.113.44",
  "pool": "pool-east-a",
  "tls": { "used": true, "version": "TLSv1.3", "verified": true },
  "dkim": [{ "domain": "customer.example", "selector": "smta1" }],
  "smtp": { "code": 421, "enhanced_status": "4.7.0", "reply": "try again later" },
  "classification": { "category": "remote-throttling", "confidence": 0.94 },
  "next_action": { "state": "deferred", "retry_at": "2026-07-12T18:45:00Z" }
}
```

3. Configure JSONL logs and retention

Write local JSONL for your high-speed write paths, then ship those logs on to your analytics or deliverability platform. Keep the active log path on fast disk, rotate it aggressively, and don't do synchronous network logging in the hot delivery path — that'll slow you down. Export failures should never be allowed to threaten queue integrity.

```
{
  "logging": {
    "connectionLogs": true,
    "deliveryAttemptLogs": true,
    "format": "jsonl",
    "path": "/var/log/signalmta",
    "retentionDays": 30,
    "redact": ["authorization", "dkim_private_key", "webhook_secret", "message_body"]
  }
}
```

4. Send events to external systems

SignalMTA can export delivery, deferral, bounce, complaint, unsubscribe, suppression, queue, TLS, auth, and policy events. Its webhooks are signed with HMAC-SHA256, carry timestamps and event ids, retry with exponential backoff, protect against replay, score endpoint health, and handle permanent failures. This format is built to be ingested by platforms like Postmastery Console and other deliverability analytics tools.

```
{
  "webhooks": {
    "endpoints": [{
      "name": "deliverability-events",
      "url": "https://analytics.example.net/signalmta/events",
      "events": ["delivered", "deferred", "bounced", "complaint", "unsubscribe", "suppressed"],
      "signingSecretRef": "env://SIGNALMTA_WEBHOOK_SECRET",
      "maxAttempts": 12,
      "retryPolicy": "exponential-backoff"
    }],
    "signing": "hmac-sha256",
    "replayProtection": true,
    "replayWindowSeconds": 300
  }
}
POST /api/webhooks/test
POST /api/webhooks/drain { "target": "deliverability-events", "limit": 100 }
GET /api/webhooks/deliverability-events/health
./bin/signalmta logs export --type delivery-attempt --since 24h --format jsonl
```

5. Use external metrics integrations when your team requires them

Day to day, the WebUI is where you'll actually watch delivery operations — start with the Dashboard, Queues, Reports, Logs, Security, and SignalAI views when you're chasing down queue age, delivery rate, deferral pressure, bounce categories, source-IP health, webhook backlog, DNS latency, TLS failures, or worker utilization. Prometheus and Grafana are there too, for teams already standardized on that stack or that need low-cardinality health signals flowing into an existing NOC/SRE workflow.

If you're exporting to Prometheus, keep the metrics low-cardinality. Labels like service, node, queue state, component, domain group, and pool are fine to use. Message id, recipient, campaign id, trace id, and raw SMTP reply text should stay in SignalMTA logs, WebUI traces, support bundles, and JSONL exports — never in a metric label.

```
# prometheus.yml
```

It starts with the following block: scrape_configs:

```
  - job_name: signalmta
    metrics_path: /api/observability/metrics
    static_configs:
      - targets: ['127.0.0.1:8080']
```

High-priority queue age — Transactional mail is waiting behind other work. — P95 > 60 seconds for 5 minutes.

Remote throttling spike — Provider capacity is being exceeded. — 4.7.x category doubles over 15 minutes.

Webhook backlog age — External analytics/support evidence may lag. — Oldest pending > 5 minutes.

DNS latency — Resolver pressure can cap throughput. — P95 lookup > 150ms sustained.

TLS required failures — Secure routes are not deliverable. — Any non-lab required-TLS failure.

6. Troubleshoot with evidence

```
./bin/signalmta logs trace --message-id msg_01J1EFGH
./bin/signalmta logs tail --type smtp-connection --domain gmail.com --source-ip 203.0.113.44
./bin/signalmta observability bundle --since 2h --domain orange.fr --include-redacted-logs
GET /api/observability/metrics?format=prometheus
GET /api/mta/insights?states=ready,deferred,held,permanent-failure
```

Security

SignalMTA locks everything down by default when it comes to relaying: nothing gets through unless you explicitly allow it. Every submitting source has to prove network permission and authenticated identity, and every operator has to work within role-scoped access plus OTP-based console protection.

SignalMTA security screenshot

Security: how relaying is locked down, rate limiting, abuse controls, and checks on who can reach management access.

Pair allowed CIDRs with SMTP AUTH, API tokens, or mTLS. Relying on CIDR restrictions alone doesn't cut it once you're running multi-tenant environments.

Turn on OTP for WebUI administrators and any operator role handling sensitive actions.

Scope every API token down to a specific tenant, stream, route, and action.

Keep DKIM private keys, webhook secrets, SMTP passwords, database credentials, and ACME material out of plain config, as secret references instead.

Audit logins, failed logins, OTP changes, token creation, config publishes, queue releases, route changes, DKIM rotation, and support bundle generation.

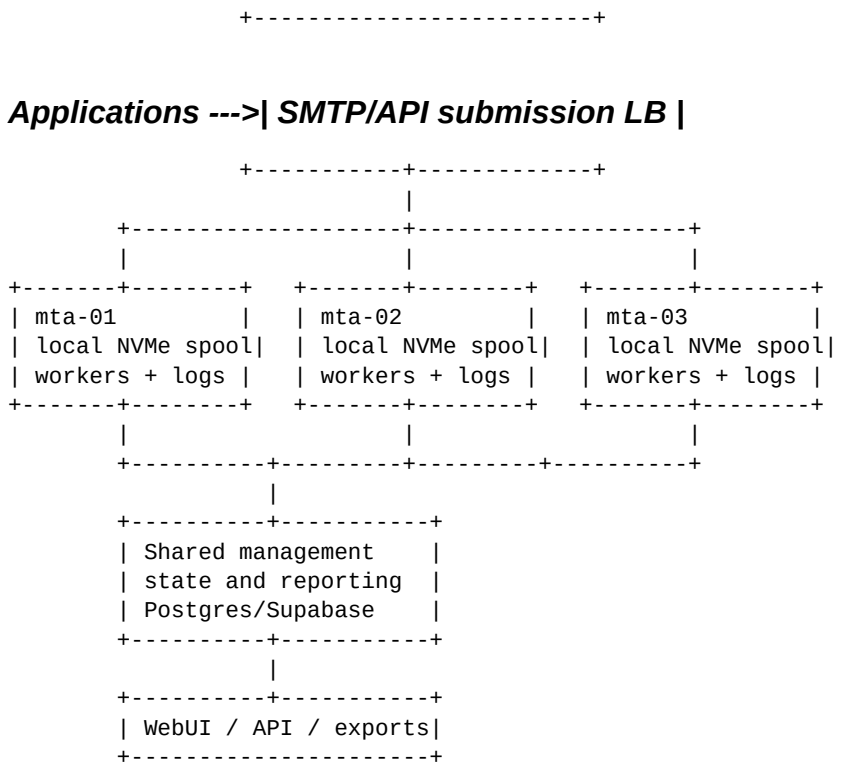
```
./bin/signalmta audit security --config /etc/signalmta/signalmta.json
./bin/signalmta user create --email ops@example.com --role engineer --mode Engineer --require-otp
./bin/signalmta token create --name stream-loader --stream transactional --actions submit,status
```

High Availability, Clustering, and Kubernetes

SignalMTA Zeta is built to run as multiple nodes with one view of the whole fleet, one shared policy, and submission routing that pays attention to node health. Each delivery node keeps its own active mail spool local, since that's what keeps delivery fast and queue integrity intact. Shared services carry policy, users, config revisions, reporting indexes, license leases, node health, and routing intelligence instead. In other words, the cluster shares its operational brain, not one giant hot-path mail queue.

Suggested single-region active/active architecture

Reach for this pattern when you're running several sending instances in one region and you want a single view across them plus the ability to pull a node out of new traffic without bringing the whole system down.



Delivery nodes — Three Zeta nodes across separate failure domains. — Lets one node drain or fail while two continue accepting new work.

Active spool — Local NVMe per node. — Do not place active spool on a shared network filesystem.

Management database — Postgres HA or Supabase-style managed Postgres. — Stores config revisions, cluster state, reporting indexes, users, and lease state.

Submission ingress — Load balancer, application router, or HAProxy layer. — Health checks must include node readiness, queue age, disk headroom, and routing eligibility.

Outbound IP model — Direct host IPs, proxy-owned egress, or Kubernetes egress gateway. — Choose one model per pool so source-IP reporting stays explainable.

Console/API — Management host or internal load-balanced service. — The WebUI reads shared state and node summaries without replacing node-local spool ownership.

Shared intelligence used for routing

Before handing a node any new submission, the cluster routing service checks it over. It looks at heartbeat freshness, license lease state, queue depth, queue age, spool headroom, filesystem latency, worker utilization, how much active SMTP session load it's carrying, DNS resolver health, TLS failure rate, destination-domain/MX acceptance rate, deferral rate, source-IP pool health, warmup phase, cooldown state, complaint rate, block-oriented replies, policy version compatibility, webhook/export backlog, and the age of the oldest pending export event.

Only send high-priority traffic to nodes that can actually meet the route, domain-auth, TLS, source-IP, queue-age, and policy requirements for that traffic. A node can be healthy overall and still be the wrong choice for one particular route — in that case the cluster should just skip it for that route, not for everything.

Bypass, drain, and maintenance modes

Active — Node can accept new submissions and deliver queued mail. — Normal operation.

Drain — Node stops receiving new submissions but continues delivering local queued mail. — OS patching, service upgrade, hardware maintenance, controlled removal.

Bypass for scope — Node remains active except for a tenant, stream, domain, MX, pool, source IP, or route. — A node has a policy mismatch, pool issue, source-IP problem, or destination-specific degradation.

Maintenance — Node is removed from new routing and expected to restart or undergo changes. — Planned downtime.

Quarantine — Node is removed from new routing because spool, logs, policy, security, or source-IP evidence needs review. — Possible corruption, source-IP mismatch, unsafe config, or unexplained failure.

WebUI routing workflow

Go to Cluster -> Nodes.

Pick the node you want.

Check its heartbeat, queue age, spool headroom, policy version, license lease, source-IP pools, and export backlog.

Click Set routing mode.

Pick Drain, Maintenance, Quarantine, or Bypass, and set the scope.

Fill in the affected scope and give an audit reason.

Check the impact preview before you publish the change.

```
./bin/signalmta cluster node drain --node mta-02 --reason "kernel patch window"
./bin/signalmta cluster node bypass --node mta-03 --scope domain:gmail.com --reason "provider-specific de
./bin/signalmta cluster routing explain --stream transactional --domain gmail.com --tenant acme
./bin/signalmta cluster node activate --node mta-02
```

Once you bypass a node, whatever mail it already accepted stays that node's responsibility until you run an explicit recovery workflow to move it. New submissions should go elsewhere. Don't just copy live spool files across nodes by hand — stick to the documented spool recovery and validation workflows.

Traffic rerouting examples

Node queue age exceeds target but disk and service health are good. — Drain new low-priority submissions to other nodes; keep high-priority reserve active if safe. — Reduces pressure without interrupting accepted mail.

Node loses access to a source-IP proxy slot. — Bypass routes that require that proxy or source IP. — Prevents source-IP mismatch and reporting confusion.

One node shows elevated TLS failures to a provider. — Bypass that provider/MX scope on the affected node and compare TLS evidence. — Keeps healthy nodes sending while isolating network or certificate issues.

Warmup cap is exhausted on one node. — Route eligible overflow traffic to an established pool on another node. — Protects new IP reputation without delaying urgent streams.

Policy revision failed validation on one node. — Keep the node active only for compatible routes or drain it until policy state is corrected. — Prevents split-brain routing behavior.

Kubernetes guidance

Only run SignalMTA on Kubernetes if your team already runs stateful workloads there. Delivery pods are stateful by nature — accepted mail has to survive restarts and pods moving to another host. Use StatefulSets for delivery nodes, persistent volumes for the spool paths, ConfigMaps for anything non-secret, Secrets for credentials, Services for API/console access, and readiness probes that check spool access, DNS, the management API, license state, queue health, and node policy.

readinessProbe (config):

```
  httpGet:
    path: /api/node/readiness
    port: 8080
  periodSeconds: 10
```

livenessProbe (config):

```
  httpGet:
    path: /api/node/liveness
    port: 8080
  periodSeconds: 30
```

volumeMounts (config):

```
  - name: spool
    mountPath: /var/lib/signalmta/spool
  - name: logs
    mountPath: /var/log/signalmta
```

Don't autoscale delivery workers off CPU alone. CPU can look perfectly fine while queue age climbs, because a provider is throttling you, DNS has gotten slow, TLS handshakes are costing more, export targets are falling behind, or you've run out of source-IP capacity. In Kubernetes, drive scaling and routing off email-specific signals instead: queue age, deferrals, acceptance rate, pool capacity, source-IP health, DNS latency, TLS failures, and export backlog.

Disaster recovery procedure

SignalMTA is meant to run stable in production, but you still need a recovery plan for host failure, storage failure, a bad config change, network outages, or a regional incident. Keep backups of your current configuration, license state, secrets, runtime state, and logs. Keep active spool on reliable local storage, and lean on cluster-level routing to stop sending new submissions to a node that's failed.

Pull the affected node out of submission routing, or mark it unavailable in cluster routing.

If the disk is still reachable, save off the spool and logs before you attempt any repair.

Restore the last known-good configuration and secret references onto the replacement hardware.

Only attach or restore spool data after filesystem checks and SignalMTA spool validation both pass.

Run spool-maintenance --check --repair-safe and look over the quarantine output before you turn delivery back on.

Only bring the node back in drain-disabled active mode once health, DNS, TLS, license, queue, and webhook checks all pass.

```
./bin/signalmta support-bundle --since 6h --redact --output ./incident-bundle.tgz
./bin/signalmta spool-maintenance --check --repair-safe --path /var/lib/signalmta/spool
./bin/signalmta config validate --config /etc/signalmta/signalmta.json
./bin/signalmta health check --full
```

Performance Tuning and Capacity Planning

Tuning SignalMTA for performance really means clearing out avoidable bottlenecks without stepping on mailbox-provider limits. Work from real evidence, not guesswork: watch queue age, accept latency, spool write latency, DNS latency, TLS negotiation time, worker utilization, provider deferrals, source-IP health, webhook/export backlog, and OS-level pressure.

Start with the right host profile

Evaluation/lab — 8 vCPU, 16 GB RAM, NVMe, 1 Gbps. — Functional workflows, domain auth, smart-sink delivery, WebUI review.

Single-node production — 16 vCPU, 32-64 GB RAM, 1 TB NVMe, 5 Gbps. — Queue age, DNS cache, TLS handshakes, delivery concurrency, logs.

High-volume node — 32+ vCPU, 64-128 GB RAM, multiple NVMe volumes, 10 Gbps. — Per-domain shaping, source-IP pools, export backlog, OS limits, resolver capacity.

Zeta cluster — 3+ delivery nodes plus HA management database. — Routing distribution, per-node queue age, source-IP ownership, failure-domain isolation.

Operating-system tuning checklist

Bump up the file descriptor limits on the SignalMTA service.

Put the spool on local NVMe and keep plenty of headroom for traffic bursts.

Keep logs on fast storage, or ship them out asynchronously from a local buffer.

Run a local caching resolver and keep an eye on its hit rate, negative cache churn, and lookup latency.

Keep system time in sync using chrony or systemd-timesyncd.

Before you push live volume, double-check outbound bandwidth, port 25 policy, reverse DNS control, and firewall state.

```
# Example systemd override
[Service]
LimitNOFILE=1048576
TasksMax=infinity
Restart=always
RestartSec=3s
```

Runtime tuning options

maxDeliveryConcurrency — Total simultaneous delivery work. — Increase gradually while DNS latency, TLS time, spool latency, and deferrals remain healthy.

maxDeliveriesPerTick — Candidate batch size per scheduler cycle. — Raise for large queues, lower if bursts create provider deferrals.

messagesPerConnection — SMTP reuse efficiency. — Increase for providers that accept reuse; lower for providers that throttle long sessions.

dnsCacheTtlMs — MX/address cache window. — Long enough to reduce lookup pressure, short enough to honor provider changes.

spool.shards — Filesystem distribution for queued messages. — Increase for very large queues to avoid directory pressure.

highPriorityReservePercent — Capacity reserved for high-priority traffic. — Raise when transactional age increases during bulk sends.

webhook.maxAttempts — Export retry durability. — Keep export retry high, but do not block delivery on remote endpoint slowness.

Performance tuning workflow

Get a baseline first: queue admission, smart-sink delivery, DNS pressure, TLS negotiation, and webhook/export overhead.

Bump concurrency up in small increments and keep watching queue age, provider deferrals, DNS latency, and CPU steal.

Don't confuse provider limits with host limits — your host can be perfectly healthy while Gmail, or some other provider, is correctly throttling a pool.

Guard your high-priority traffic with scheduler reserve and queue-age alerts.

Pull heavy analytics and external exports off the delivery hot path.

Hold on to your tuning evidence so you can compare future changes back to a known-good baseline.

```
./bin/signalmta performance-qualification \
  --edition zeta \
  --intended-mph 1000000 \
  --cores 32 --memory-gb 64 --network-gbps 10 --nvme-gb 2000 --source-ips 32

./bin/signalmta dns-pressure-check --lookups 100000 --unique-domains 10000
./bin/signalmta throughput-readiness --sustained-report ./soak.json --dns-report ./dns.json
./bin/signalmta queue summary --group-by priority,domain,pool,state
```

When performance falls short

Queue age rises only for one domain — Provider-specific throttle or block. — Inspect domain drilldown, 4.7.x rate, source IP, and retry schedule.

Queue age rises globally — Host, spool, DNS, worker, or export bottleneck. — Check spool latency, DNS latency, worker saturation, CPU steal, and webhook backlog.

DNS latency increases — Resolver saturation or cache misconfiguration. — Increase resolver capacity, tune TTLs, reduce unique-domain pressure.

TLS time dominates attempts — Handshake volume, remote MX behavior, or strict route policy. — Review connection reuse, TLS policy, and provider-specific caps.

Export backlog grows — External endpoint is slow or unavailable. — Let local delivery continue, drain webhooks, and review endpoint health.

How to Operate Each Major Feature

This section is for when you need to actually do something, not just read about what a feature is. Every workflow below walks through the WebUI path, the minimum config you need, how to check your work, the equivalent CLI/API calls, what evidence to look at, and the mistakes people make most often.

1. Set up SMTP submission listeners

Go to Security -> Relay Control first and check that the default policy is deny-by-default.

Go to Configuration -> Listeners and create one listener per submission role. Typically that's 587 for authenticated customer or app submission, 2525 for private load-balancer or test traffic, and 465 only if a sender specifically needs implicit TLS.

On any public or customer-facing listener, require STARTTLS before AUTH, and point it at a valid certificate reference or an ACME-managed certificate.

Only add relay CIDRs for networks you actually trust: application servers, VPN ranges, private load balancers, customer networks. Never open relay with 0.0.0.0/0.

Under Security -> SMTP AUTH, create your AUTH identities, then bind each one to a tenant, stream, route, and the X-Signal controls it's allowed to use.

Validate the config, publish it, and test with swaks. Then check Logs -> Admission for the listener id, auth identity, source IP, tenant, stream, and queue result to confirm it landed the way you expected.

*swaks --server mta.example.com --port 587 --tls *

```
--auth LOGIN --auth-user acme-transactional --auth-password "$SMTP_PASS" \
--from bounce+msg_01J125@bounces.example.com \
--to user@example.net \
--header "X-Signal-Stream: transactional" \
--header "X-Signal-Priority: high" \
--body "Submission test"
```

STARTTLS negotiated before AUTH — Logs -> SMTP connection — tls_used=true, authenticated identity present.

Relay permission — Security -> Relay Control — Source CIDR and SMTP AUTH identity both match.

Queue admission — Queues and Logs -> Admission — Message is accepted once and assigned to the expected stream.

2. Create API injection tokens and submit mail by API

Go to Security -> API Tokens and create a token scoped to whichever tenant and stream will be submitting mail.

Grant only what the application actually needs, usually submit and maybe status. Don't hand out administrator tokens for injection.

Configure payload limits, which From domains are allowed, which route/pool requests are allowed, and your idempotency policy.

Submit a message with an Idempotency-Key, then retry that same request and check that SignalMTA hands back the identical accepted message id.

Check Logs -> Admission for the token name, tenant, stream, idempotency key, and queue lane.

```
curl -X POST https://mta.example.com/api/mta/enqueue \
  -H "Authorization: Bearer $SIGNALMTA_TOKEN" \
  -H "Content-Type: application/json" \
  -H "Idempotency-Key: receipt-98271-v1" \
  -d '{"tenant":"acme","stream":"transactional","priority":"high",
       "mailFrom":"bounce+msg_01J125@bounces.example.com",
       "recipients":["user@example.net"],
       "headers":{"From":"Acme <receipts@example.com>","Subject":"Receipt"},
       "mime":"From: Acme <receipts@example.com>\r\nSubject: Receipt\r\n\r\nThank you."}'
```

If a caller times out, it should just retry with the same idempotency key. If it instead reuses the key but changes the content, SignalMTA rejects the request so you don't end up injecting duplicate or mismatched mail.

3. Build a submission-flow rule

Go to Configuration -> Submission Flow and create a rule with a name that actually tells you what it does, something like acme transactional route.

Pick your match conditions: tenant, stream, SMTP AUTH user, API token, source network, From domain, tag, campaign, or SSL decision.

Pick the result: virtual instance, queue lane, priority, route, source-IP pool, DKIM selector, TLS policy, and fallback behavior.

Decide what happens if a requested control isn't permitted: reject it, ignore it, park it, or downgrade its priority.

Run Explain rule before you publish. Then submit a test message and confirm the route it took in message drilldown.

```
{
  "submissionFlow": {
    "rules": [{
      "name": "acme transactional route",
      "match": { "tenant": "acme", "stream": "transactional" },
      "set": {
        "priority": "high",
        "queueLane": "transactional",
        "route": "direct-us-east",
        "pool": "pool-acme-dedicated",
        "sendingDomain": "example.com"
```

```
      },
      "rejectUnauthorizedControls": true
    }]
  }
}
```

4. Create and verify a sending domain with DKIM

Go to Domains -> Add sending domain.

Fill in the Header From domain, the return-path domain, the tenant/stream binding, and the default route or pool.

Pick Generate selector if this is a new domain, or Attach existing selector if you already have key material for it.

Store private keys only as secret references. Never paste a DKIM private key into a ticket, a screenshot, or a support bundle.

Click Generate DNS records, then publish the DKIM TXT record along with SPF guidance, DMARC guidance, and the return-path records.

Keep clicking Verify DNS until selector visibility, key parsing, return-path reachability, SPF syntax, DMARC syntax, and alignment all pass.

Send a test message and open the delivery attempt. Check the selector, signing domain, aligned From domain, and double-signing evidence if you've got that enabled.

```
./bin/signalmta domain-auth-plan \
  --domain example.com \
  --selector smta1 \
  --bounce-domain bounces.example.com \
  --dmarc-policy quarantine
```

5. Configure queue priorities and protect urgent mail

Go to Queues -> Scheduler.

Set up queue lanes for transactional, standard, bulk, warmup, deferred, and provider-relay traffic.

Set a high priority reserve so transactional mail keeps its own worker/DNS capacity even while a bulk send is running.

Set maximum candidate windows so a single oversized stream can't hog the scheduler's reads.

Send mixed high-priority and bulk test traffic through and confirm high-priority queue age stays low while the bulk mail keeps moving.

```
{
  "queueScheduler": {
    "highPriorityReservePercent": 35,
    "candidateWindowMultiplier": 20,
```

```
    "maxCandidates": 10000,
    "maxPerPoolPerTick": 0
  }
}
```

If high-priority queue age climbs during bulk sends, raise the reserve, cut bulk caps, break the oversized stream into fairer lanes, or move noisy traffic to its own dedicated virtual instance.

6. Use traffic shaping and retry rules

Open Traffic Shaping and pick the smallest scope that actually explains the problem: recipient domain, MX rollup, source-IP pool, source IP, stream, tenant, route, or error category.

Start conservative: set caps for messages/minute, concurrent connections, messages/connection, retry windows, and cooldown behavior.

Turn on response-driven adjustments, but keep guardrails on them: a maximum reduction, a minimum floor, a cooldown duration, and approval requirements.

Map error classes to the right retry behavior. Greylisting, for instance, should retry quickly with jitter; remote throttling should slow down that scope; block-oriented replies should hold for a deliverability review.

Run Explain policy before publishing. Once it's live, check queue age, deferral rate, remote acceptance, and the SignalAI evidence for that scope.

```
./bin/signalmta shaping explain --domain gmail.com --pool warmup-a --source-ip 203.0.113.44
./bin/signalmta config publish --config ./candidate.json --reason "Reduce Gmail warmup cap after 4.7.x bu
```

7. Create automated IP warmup

Go to Configuration -> Warmup and click Create plan.

Pick the source IP or leased range, the pool, a proxy slot if you're using one, the tenant, the stream, a start date, an initial cap, a target cap, and a warm overflow pool.

Group recipients the way mailbox providers actually break down: Gmail, Microsoft, Yahoo/AOL, Apple, regional providers, and long-tail domains.

Set guardrails on complaint rate, permanent failure rate, block-oriented replies, provider deferrals, authentication failures, and DNS/rDNS readiness.

Publish the plan in automatic mode with operator override still available. After every DNS or reputation fix, hit Evaluate now.

Before you ramp traffic up, check cap history, the next scheduled decision, any paused groupings, overflow routing, and the complaint/bounce categories.

```
POST /api/ip-warmup/plan
{
  "sourceIp": "203.0.113.44",
```

```
  "pool": "warmup-us-a",
  "startAt": "2026-07-20T14:00:00Z",
  "initialDailyCap": 500,
  "targetDailyCap": 250000,
  "growthPercentPerDay": 25,
  "overflowPool": "pool-established-us"
}
```

8. Investigate bounces, complaints, and unsubscribes

Go to Reports -> Bounce categories and filter by tenant, stream, recipient domain, MX, pool, source IP, or campaign/tag.

Open one event that represents the pattern and check its enhanced status code, the raw SMTP reply or DSN, localized text, category, confidence, source IP, route, and retry decision.

For asynchronous DSNs, confirm that VERP or the fallback correlation actually tied it back to the original message, attempt, route, source IP, and remote acceptance event.

For complaints, open Feedback Loops and check the ARF evidence, Feedback-ID, original headers, suppression outcome, and export event.

Don't lump complaints in with ordinary bounces. Complaint suppressions, unsubscribe suppressions, and permanent-failure suppressions all need to stay separately reportable.

```
./bin/signalmta bounces classify --sample ./dsn.eml --explain
./bin/signalmta reports bounce-drilldown --stream marketing-us --domain orange.fr --since 2h
GET /api/reports/bounces?stream=marketing-us&category=block_oriented&includeEvidence=true
```

9. Configure webhooks and external exports

Go to Observability -> Webhooks and create a separate endpoint for each destination system.

Choose which event types to send: delivery attempt, deferred, bounce, complaint, unsubscribe, suppression, policy, TLS, auth, queue, and out-of-band DSN.

Turn on HMAC-SHA256 signatures, set your timestamp tolerance, retry policy, endpoint health checks, and pending event limits.

Fire a test event and confirm the receiving system validates the signature, then replay one real event from Logs.

Keep an eye on pending count, oldest pending age, retry count, last HTTP status, signature failures, and whether the endpoint has been marked failed.

```
{
  "observability": {
    "webhooks": [{
      "id": "postmastery-console",
      "url": "https://collector.example.com/signalmta",
      "events": ["delivery_attempt", "bounce", "complaint", "unsubscribe"],
      "signing": "hmac-sha256",
```

```
      "secretRef": "env://POSTMASTERY_WEBHOOK_SECRET",
      "maxAttempts": 12
    }]
  }
}
```

10. Use WebUI logs first, then external observability if needed

Reach for Logs -> Trace when you need to see exactly what happened to one message from submission to final outcome.

Reach for Reports when you need grouped answers instead, broken out by tenant, stream, recipient domain, pool, source IP, category, or time window.

For day-to-day troubleshooting, the WebUI's dashboard, queue, report, and SignalAI views should cover you. If your org already runs Prometheus or Grafana, export the low-cardinality health signals there instead: queue depth, queue age, worker utilization, DNS latency, webhook backlog, remote acceptance rate, and deferral rate.

Keep recipient addresses, message ids, trace ids, raw SMTP replies, campaign ids, and custom header values out of Prometheus labels entirely.

When you open a support case, send a redacted support bundle rather than just screenshots.

```
./bin/signalmta logs trace --message-id msg_01J1EFGH
./bin/signalmta support-bundle --since 24h --redact --output ./support-bundle.tgz
curl http://127.0.0.1:4780/api/metrics/prometheus
```

11. Manage IP pools, dedicated IPs, proxies, and IPXO inventory

Go to Configuration -> IP Pools and organize pools by what they're actually used for: transactional dedicated, shared marketing, warmup, cooldown, regional, provider-relay overflow, or customer-dedicated.

When adding source IPs, record owner, tenancy, rDNS, SPF include, warmup state, proxy id, lease metadata, and which streams are allowed to use it.

If you're running HAProxy or custom egress, go to Outbound Proxies and check the observed source IP, health check, TLS capability, and pool mapping.

For IPXO, go to IP Inventory -> IPXO, connect the API token by secret reference, sync your inventory, approve ranges, split /24 blocks into your operating pools, and run them through warmup before you send production traffic.

Confirm your reports can actually tell you which tenant, stream, route, pool, proxy, leased range, and source IP were involved when something goes wrong.

```
./bin/signalmta ipxo sync --dry-run
./bin/signalmta ipxo split --range 203.0.113.0/24 --into warmup-a=/28,transactional-a=/28,marketing-a=/27
./bin/signalmta proxy validate --proxy egress-east-a --pool marketing-a
./bin/signalmta ip validate --ip 203.0.113.44 --checks rdns,spf,proxy,warmup
```

12. Use Signal Scripting Language safely

Open SSL Policies and pick the hook: injection, routing, pre-delivery, SMTP reply, post-delivery, bounce, complaint, or webhook.

Write the narrowest policy you can get away with. Prefer checks scoped to stream, tenant, domain, pool, and error class over anything global.

Run validation and a dry-run against sample messages before you publish.

Look at the decision trace afterward. It should tell you exactly which rule fired, what action it took, and what metadata it changed.

Publish with an audit reason attached, and keep a rollback path ready.

```
hook injection(message) {
  require_header "List-Unsubscribe" when message.stream == "marketing";
  set_priority "high" when message.stream == "transactional";
}

hook smtp_reply(reply, context) {
  if reply.status ~= "4.7." {
    throttle domain -35% for 30m;
    retry after "15m,45m,2h";
  }
}
```

13. Operate SignalAI advisories

Open SignalAI and go through advisories by severity, affected scope, confidence, evidence, and recommended action.

Click into an advisory to see the metrics, logs, reports, and recent config changes backing it up.

Only approve actions that fit your operating policy. Guarded actions like route changes, pool holds, throttle reductions, and anything touching suppression should need a human to sign off.

Once approved, check the generated config diff and reload plan before it goes live.

Watch whether queue age, deferral rate, bounce category, complaint rate, or webhook backlog actually improves after the change lands.

14. Add users, roles, OTP, and audit controls

Go to Security -> Users and create named accounts for each person. Never share an administrator account.

Give each user the right UI mode: CTO for executive health/cost/license footprint, Engineer for operations/configuration, Deliverability for domains/bounces/warmup/reputation work.

Require OTP for administrators and any sensitive operator role, and store the recovery codes somewhere secure.

Create API tokens as their own thing, separate from human users, and scope each one by tenant, stream, action, and expiry.

Check the audit logs after every config publish, queue release, route change, DKIM rotation, user change, and support bundle export.

15. Run production readiness before live sending

Run host qualification for the workload and edition you're actually deploying.

Run domain-auth checks against every sending domain and return-path domain.

Run submission tests against every listener, SMTP AUTH identity, API token, and message stream.

Run a smart-sink or provider simulation to check queue admission, retry behavior, bounce classification, webhook export, and the support bundle workflow.

Run go-live preflight, and treat anything it flags as a blocker as actual required work, not just advisory copy.

```
./bin/signalmta qualify-host --profile alpha-250k --target-mph 250000 --report-dir /var/log/signalmta/rea
./bin/signalmta audit:golive --config /etc/signalmta/signalmta.json --public-smtp
./bin/signalmta support-bundle --since 24h --redact --output ./preflight-bundle.tgz
```

Complete Feature Operation Reference

This section walks through how each SignalMTA functional area actually behaves once it's running in production: what you as the operator can configure, which WebUI view shows it, which logs prove what happened, and what to keep an eye on while real traffic is flowing.

Think of this as the map to the product manual. Every feature here is built to answer four questions: what does it do, how does it decide, what can you tune, and how do you prove the outcome afterward?

1. SMTP receiving service

The SMTP receiving service listens on the addresses and ports you've configured, advertises ESMTP capabilities, negotiates inbound STARTTLS when it's turned on, authenticates SMTP AUTH users, checks the allowed relay CIDRs, validates message size, applies submission-flow policy, and writes accepted mail to the durable spool. It turns away unsafe traffic before queue admission, so mail you don't want never eats into spool, retry, source-IP, or reputation capacity.

realMta.smtpHost, smtpPort — Controls listener binding for inbound SMTP. — Binding publicly before relay controls are ready creates open-relay risk.

realMta.requireAuth — Requires SMTP AUTH or equivalent authenticated source. — Keep enabled for all production public-facing listeners.

realMta.allowedRelayCidrs — Limits which networks can submit mail. — CIDR is not identity; pair it with SMTP AUTH, API token, or mTLS.

realMta.maxMessageSizeMb — Rejects oversized messages before spool write. — Large MIME payloads reduce throughput and increase spool pressure.

realMta.inboundTls.requireTlsForAuth — Prevents credentials from being sent before STARTTLS. — Disabling this should fail production safety audit.

2. API injection

The API injection path takes structured message submissions from approved applications. It handles bearer-token identity, tenant/stream scope, idempotency keys, custom metadata, X-header preservation policy, and the same submission-flow decisions SMTP injection uses. If a caller retries after a timeout with the same idempotency key, SignalMTA hands back the original accepted message identity instead of creating a duplicate.

```
POST /api/mta/enqueue
Authorization: Bearer $SIGNALMTA_TOKEN
Idempotency-Key: order-889172
Content-Type: application/json

{
  "tenant": "acme",
  "stream": "transactional",
  "priority": "high",
  "from": "receipts@example.com",
  "to": ["buyer@example.net"],
```

```
  "headers": { "X-Customer-Trace": "order-889172" },
  "mime": "..."
}
```

3. Submission flow control

Submission flow works out where each accepted message lands: virtual instance, queue lane, IP pool, source IP intent, sending domain, DKIM selector, priority, and cap. Rules can match on tenant, API token, SMTP AUTH user, source network, tag, campaign, stream, From domain, custom domain, or an SSL script decision. The route it resolves gets stored on message metadata and shows up in exported delivery attempts.

Virtual instance — Logical lane for node, pod, policy boundary, route group, or source-IP family. — route.virtualInstance in admission and attempt logs.

Queue lane — Maps traffic into transactional, standard, bulk, provider-relay, warmup, or custom lanes. — Queue dashboard by lane and priority.

IP pool/source IP — Selects allowed pool and optional source IP or proxy egress intent. — Attempt log source IP, pool, lease id, proxy.

Overflow — Park, defer, route to warm pool, route to provider, or reject when caps are exceeded. — Route-policy deferral with plain-English explanation.

SignalMTA configuration WebUI view

This is the real WebUI configuration view for domains, routing, the queue scheduler, warmup, webhooks, and validation controls.

4. Virtual instances

Virtual instances let you isolate traffic without spinning up separate operating-system servers. A virtual instance might represent a customer lane, a performance lane, a Kubernetes pod family, a bare-metal group, a cloud region, or a cooling lane. Operators use them to stop transactional traffic, marketing traffic, warmup traffic, provider-relay overflow, and reputation-sensitive cohorts from interfering with each other.

```
{
  "submissionFlow": {
    "virtualInstances": [
      { "id": "vi-transactional-east", "queueLane": "transactional", "defaultPool": "pool-east-a" },
      { "id": "vi-marketing-cooldown", "queueLane": "bulk", "defaultPool": "pool-cooldown" }
    ]
  }
}
```

5. Queue and spool engine

The spool is where reliability actually lives. SignalMTA writes every accepted message and its metadata to durable storage before it even attempts delivery. The queue engine keeps ready mail, deferred mail, held mail, delivered mail, permanent failures, and corrupt/quarantine states separate. Metadata tracked includes tenant, stream, route, priority, attempt count, retry plan, queue lane, recipient domain, source pool, custom trace fields, and recent attempt history.

Queue integrity is backed by checksums, fsync batching, checkpoint intervals, restart recovery scans, corrupt-message quarantine, admission limits by message count and by bytes, redrive guardrails, and spool maintenance checks.

spool.shards — Distributes queue files to reduce directory pressure. — 64 shards for production config.

spool.checkpointIntervalMs — Controls metadata checkpoint cadence. — 500ms baseline.

realMta.maxQueuedMessages — Prevents unbounded acceptance. — Finite positive value required.

queueIntegrity.quarantineCorruptMessages — Moves unreadable/corrupt items out of active lanes. — Enabled.

6. Priority scheduler

The scheduler picks candidates out of the ready queues, then applies priority reserve, per-recipient-domain/MX shaping, pool/source-IP caps, retry timing, warmup state, and route health. The high-priority reserve keeps urgent traffic safe from bulk streams, and candidate windows stop one giant stream from starving smaller queues.

```
"queueScheduler": {
  "candidateWindowMultiplier": 20,
  "maxCandidates": 10000,
  "maxPerDomainPerTick": 0,
  "maxPerPoolPerTick": 0,
  "maxPerSourceIpPerTick": 0,
  "highPriorityReservePercent": 35
}
```

7. Delivery workers

Delivery workers pull ready messages off the queue, resolve the recipient domain's MX records, open SMTP sessions, negotiate STARTTLS, apply DKIM signing, send the message, classify the reply, write attempt logs, update queue state, and schedule a retry or final disposition. Worker concurrency is capped globally, and further constrained by domain, pool, source IP, and route policy.

maxDeliveryConcurrency — Maximum simultaneous delivery workers. — Raise only after DNS, remote limits, and spool latency stay healthy.

maxDeliveriesPerTick — Batch of delivery candidates per scheduler cycle. — Use with candidate windows to prevent queue bursts.

outboundSmtpPort — Remote SMTP port, normally 25. — Use provider-specific connectors for 587 relay routes.

smtpTimeoutMs — Socket and SMTP phase timeout. — Too low creates false connection failures; too high ties up workers.

8. DNS cache and lookup pressure

DNS matters here as a performance subsystem. SignalMTA caches MX and address lookups with bounded positive and negative TTLs, tracks resolver latency, logs lookup failures, and stops unbounded concurrency from overwhelming your local resolvers. Run a local caching resolver near the MTA, and watch cache hit rate, NXDOMAIN/timeout rate, and lookup latency during load tests.

```
"realMta": { "dnsCacheTtlMs": 300000 },
"dns": {
  "cache_ttl_seconds": 300,
  "negative_ttl_seconds": 60,
  "max_concurrent_lookups": 2000
}
```

9. TLS security and evidence

Outbound TLS has three modes: disabled, opportunistic, and required. Opportunistic mode uses STARTTLS whenever it's available. Required mode defers delivery if STARTTLS isn't available or certificate verification fails. You can override TLS policy per route or domain, and a strict MTA-STS or configured DANE/TLSA policy can force required mode.

Attempt logs record TLS mode, whether TLS actually got used, TLS version, cipher where available, SNI, verification result, the remote MX, and the reason for any TLS failure. The WebUI surfaces this evidence so you don't mistake a TLS problem for provider throttling or a recipient failure.

```
"tlsPolicy": {
  "minVersion": "TLSv1.2",
  "rejectUnauthorized": true,
  "requireTlsForAuth": true,
  "mtaSts": { "enabled": true, "cacheTtlMs": 86400000, "enforceMode": "testing" },
  "dane": { "enabled": false, "requireDnssecValidation": true }
}
```

10. ACME and certificate automation

When ACME is enabled, SignalMTA manages the certificate inventory for inbound SMTP and WebUI/API. The ACME plan tracks the directory URL, account key reference, contact email, challenge type, challenge ingress, certificate references, key references, renewal window, issuer, expiration, and renewal logs. You get warnings before expiration, and you can fall back to manual certificate references if ACME isn't available.

11. Sending domains

A sending domain ties together the visible From identity, return-path domain, DKIM selectors, SPF/DMARC guidance, how TLS is set up, bounce correlation, and stream permissions. The domain

verification flow checks DNS records, selector visibility, return-path MX, DMARC policy, alignment, and bounce handling before you scale up a stream on it.

```
{
  "authentication": {
    "sendingDomains": [{
      "domain": "example.com",
      "tenant": "acme",
      "returnPathDomain": "bounces.example.com",
      "dkimSelectors": ["sig1", "platform1"],
      "defaultPool": "pool-east-a"
    }]
  }
}
```

12. DKIM, Ed25519, double DKIM, and DKIM2

DKIM signing happens during delivery, after route and sending-domain selection. SignalMTA supports RSA SHA-256 selectors, Ed25519 selectors, selector routing per tenant or per sending domain, a rotation workflow, signing traces, and double DKIM signing. Double signing attaches both the customer-domain signature and the service-provider/platform signature to the same message, and records both identities in the attempt logs. DKIM2 stays behind explicit configuration and compatibility checks.

Selector routing — Selects DKIM key by domain, tenant, stream, or SSL decision. — Attempt log selector list.

Double signing — Signs once for customer identity and once for platform identity. — Two DKIM identities on attempt record.

Rotation — Publish next selector, verify DNS, cut over, retire old selector. — Domain-auth plan and audit log.

Failure handling — Blocks or defers when required signing material is missing. — Auth category with operator explanation.

13. Smart-sink and provider simulation

Smart-sink mode lets teams exercise queue admission, delivery loops, retry classification, webhook export, and suppression logic without actually sending to public mailbox providers. Production config ships with smart-sink rules that bounce .invalid domains and defer domains containing tempfail. Run smart-sink before you touch public SMTP, so the runtime path gets tested safely first.

14. Bounce processor

The bounce processor parses direct SMTP replies, RFC-shaped DSNs, multipart/report messages, message/delivery-status parts, enhanced status codes, provider text, non-English diagnostics, and double-byte content. Classification puts standards first: enhanced status code and DSN fields count as primary evidence, provider text and phrase maps are secondary evidence, and the raw diagnostics get kept around for support to review.

Bad recipient — Matches 5.1.x, unknown user, disabled mailbox, invalid address. — Suppress recipient and stop retry.

Remote throttling — Matches 4.7.x, too many connections/messages, try later. — Retry with throttle and jitter.

Auth/policy failure — Matches 5.7.x, SPF/DKIM/DMARC/TLS policy text. — Pause stream or require authentication review.

Block-oriented — Detects reputation block, deny list, IP/domain block language. — Hold affected scope and alert deliverability.

Localized/regional — Uses UTF-8 preservation, language signals, provider profile, region hints. — Classify with confidence and preserve raw text.

15. Out-of-band bounces and VERP

Out-of-band bounces show up after a remote system has already accepted the message. SignalMTA uses VERP-style tokenized return paths to trace a DSN back to the tenant, stream, message id, recipient, attempt, source IP, route, and original SMTP acceptance. When VERP isn't there, it falls back to Message-ID, X-Signal trace headers, provider queue ids, and DSN fields.

```
POST /api/mta/out-of-band-bounce
{
  "returnPath": "b+tenant.stream.messageid.1@bounces.example.test",
  "recipient": "user@example.test",
  "action": "failed",
  "enhancedStatus": "5.1.1",
  "diagnosticCode": "smtp; 550 5.1.1 mailbox disabled"
}
```

16. Feedback loops and unsubscribe processing

Feedback loop reports get parsed as complaint events, not treated as ordinary bounces. SignalMTA accepts full ARF multipart/report messages or structured feedback fields at POST /api/mta/feedback-report. It reads message/feedback-report fields, the original message headers, Feedback-ID, X-SignalMTA-Message-ID, X-SignalMTA-Trace-ID, tenant, stream, source IP, and provider evidence. A complaint report suppresses the recipient whenever the provider identifies one. When the provider redacts the recipient instead, SignalMTA still logs the complaint, ties it back to the message or stream where it can, and exports the event so reporting and SignalAI can show complaint pressure at the stream level.

```
{
  "bounceProcessor": {
    "feedbackLoops": {
      "enabled": true,
      "arf": true,
      "parseOriginalMessageHeaders": true,
      "trackingHeaders": true,
      "feedbackIdFormat": "{tenant}:{stream}:{tag}:{message_id}",
      "redactedRecipientPolicy": "record-correlated-complaint-without-recipient-suppression",
      "ingestionEndpoint": "/api/mta/feedback-report"
```

```
    }
  }
}
```

One-click unsubscribe requests get classified separately from complaints, and they suppress the recipient only from that specific stream or list. Keep complaint events for abuse handling, unsubscribe events for consent handling, and bounce events for delivery-failure handling — mixing these signals up makes your reporting noisy and can trigger the wrong operational response.

17. Suppression model

Suppression checks happen before queue admission. Permanent recipient failures, complaints, and unsubscribes each create a suppression record carrying tenant, stream, recipient hash, domain, source event, category, explanation, and audit metadata. Controlled operator workflows can bypass suppression with an explicit header, but every bypass gets logged.

18. Webhook export

Webhooks export delivered, deferred, bounced, complaint, suppressed, policy, TLS, auth, queue, and out-of-band DSN events. Delivery relies on HMAC-SHA256 signatures, replay protection, retry with exponential backoff, endpoint health tracking, pending queues, and a failed-endpoint state once retries are exhausted. Operators can drain, simulate, reset health, and check pending counts.

```
POST /api/webhooks/drain
{
  "target": "delivery-events",
  "limit": 50,
  "simulate": false
}
```

19. SMTP connection logs and delivery attempt logs

Connection logs cover the transport session itself: node, cluster, connection id, trace id, proxy, source IP, remote peer, MX, SMTP phase, TLS version, result, and latency. Delivery attempt logs cover the message outcome: tenant, stream, campaign, tag, idempotency key, recipient domain, MX, attempt number, source IP, route pool, leased range, warmup phase, SMTP code, enhanced status, classification, retry time, and explanation.

20. Observability, WebUI, Prometheus, and Grafana

SignalMTA's WebUI is where you'll want to watch things by default. It's built specifically for MTA operators and deliverability teams, so it pulls queue state, delivery attempts, bounce intelligence, source-IP behavior, domain/MX pressure, config changes, security setup, webhook health, and SignalAI advisories into one place. If you already run a monitoring stack, the observability API is there for you too, including Prometheus text exposition, metric samples, health checks, and export bundles.

Prometheus and Grafana support is optional — reach for it when your organization already has NOC/SRE dashboards or alert-routing practices that should fold in SignalMTA's health signals. Keep Prometheus metrics to low-cardinality labels like service, cluster, node, environment, queue state, domain group, and component. High-cardinality values such as message id, recipient, campaign id, and trace id belong in WebUI traces, JSONL logs, support bundles, and event exports instead.

SignalMTA reports WebUI view

This is the real WebUI reports view for category trends, domain drilldown, source-IP performance, and the context behind operator actions.

Observability is split into three layers so high-volume delivery never has to wait on a slow analytics target. First, the runtime writes local structured logs and counters right on the hot path. Second, the export service batches and signs events for webhooks, files, syslog, Kafka, AMQP, or a deliverability platform. Third, the WebUI reads back summarized operational state for live troubleshooting, while Prometheus/Grafana can pull the same low-cardinality health data if you need an external monitoring stack.

Queue age and depth — WebUI Queues, Prometheus, JSONL queue events. — Alert on high-priority queue age before alerting on CPU.

Delivery attempts — WebUI Reports, delivery-attempt JSONL, webhook exports. — Join tenant, stream, domain, MX, source IP, TLS, DKIM, SMTP reply, category, and retry decision.

SMTP connections — Logs view, connection JSONL, support bundle. — Inspect HELO, source IP, proxy id, TLS handshake, command phase, remote close, and latency.

DNS behavior — DNS metrics, resolver cache stats, SignalAI advisories. — Find resolver pressure, cache misses, slow MX lookups, and DNSSEC/MTA-STS lookups that affect throughput.

Webhook/export health — Reports -> Export health, Webhooks settings, metrics. — Track pending count, oldest pending age, retry rate, endpoint status, signature failures, and failed webhook events.

Security and config changes — Security view, audit logs, config revision history. — Prove who changed relay permissions, routes, pools, users, tokens, webhooks, DKIM, or TLS policy.

```
{
  "observability": {
    "metrics": {
      "enabled": true,
      "listen": "127.0.0.1:8080",
      "path": "/api/observability/metrics",
      "format": "prometheus",
      "domainLabelMode": "grouped"
    },
    "logs": {
      "format": "jsonl",
      "path": "/var/log/signalmta",
      "connection": true,
      "deliveryAttempt": true,
      "queueTransition": true,
      "audit": true,
      "redactBodies": true
    },
```

```
    "exports": {
      "asyncOnly": true,
      "maxBatchSize": 500,
      "oldestPendingAlertSeconds": 300
    }
  }
}
```

If you do bring in Grafana, keep it focused on operational health rather than trying to replace the SignalMTA WebUI. Good external panels include delivery volume, accepted volume, delivered volume, queue age by priority, ready/deferred/held/permanent-failure counts, remote-throttling rate, bounce category rate, complaint rate, DNS latency, TLS failure rate, source-IP pool health, webhook backlog, and worker utilization. Don't build a dashboard that's just CPU, memory, and disk — email delivery trouble usually shows up first as queue age, remote deferrals, DNS pressure, or export lag.

```
# Prometheus scrape example
```

Here's the scrape_configs: block to drop into your Prometheus setup.

```
  - job_name: signalmta
    metrics_path: /api/observability/metrics
    scheme: http
    static_configs:
      - targets:
          - mta-01.internal:8080
          - mta-02.internal:8080
          - mta-03.internal:8080
```

21. WebUI reporting

Reporting is built around the questions you actually ask: what's moving, what's blocked, which domains are changing, which streams are risky, which source IPs are underperforming, and whether exports are healthy. Reports pull together delivery attempts, queue transitions, bounce categories, complaint/unsubscribe events, TLS evidence, DNS metrics, warmup state, and SignalAI recommendations.

The dashboard shows delivery rate, queue age, the bounce/complaint trend, and active alerts.

Reports cover category trends, the domain leaderboard, source-IP pool performance, and export health.

Domain drilldown surfaces MX-level throttles, deferrals, TLS, DNS, provider replies, and the retry curve.

Stream drilldown covers customer/campaign/tag behavior, custom X-header joins, and compliance events.

SignalAI adds the anomaly explanation, affected scope, recommended action, and approval state.

22. WebUI configuration

Configuration screens are guided forms sitting on top of the same runtime config model. From there operators can edit domains, DKIM, TLS, relay sources, SMTP AUTH, API tokens, routes, virtual instances, IP pools, source IPs, the queue scheduler, warmup plans, webhooks, logging, export targets, users, OTP, RBAC, and cluster nodes. Every publish generates a config revision, a validation result, a diff, the actor, the reason, the reload impact, and a rollback target.

For day-to-day operations, the WebUI publishes through POST /api/runtime-control/publish. That's how an Engineer can adjust a destination throttle, a Deliverability operator can rotate a DKIM selector, or an operator can add a webhook endpoint — all without hand-editing JSON. SignalMTA turns the guided action into a config patch, checks its scope and safety, writes a revision, logs an audit event, and hot-reloads whatever runtime-safe settings changed.

Runtime-control publish covers throttle rules, retry rules, sending-domain authentication, bounce categories, feedback-loop and VERP behavior, webhook endpoints, IP pools, outbound proxies, warmup schedules and overrides, cluster routing policy, routes, and submission-flow rules. CTO mode stays observational; Engineer and Deliverability users can only publish within their assigned RBAC and tenant/stream scope.

23. SignalAI

SignalAI keeps watch over queue anomalies, provider deferrals, bounce spikes, authentication/TLS issues, DNS pressure, warmup drift, webhook backlog, cluster routing imbalance, license footprint risks, and source-IP reputation signals. It explains the affected scope, the likely cause, a confidence level, a recommended action, and the evidence behind that recommendation. Advisory mode is the default; guarded automatic actions need an explicit policy first.

SignalAI can run in rules-only mode, or lean on a self-hosted open-source LLM for richer explanations. The recommended default model is mistralai/Mistral-Small-3.1-24B-Instruct-2503 behind a local OpenAI-compatible endpoint such as vLLM. This is off by default, uses low-temperature JSON output, and falls back to rules-only recommendations if the model is unavailable, too slow, or returns invalid JSON.

```
{
  "signalAI": {
    "enabled": true,
    "mode": "advisory",
    "llm": {
      "enabled": true,
      "provider": "openai-compatible",
      "model": "mistralai/Mistral-Small-3.1-24B-Instruct-2503",
      "endpoint": "http://127.0.0.1:8000/v1/chat/completions",
      "deployment": "local-vllm",
      "fallbackMode": "rules-only",
      "allowExternalNetwork": false,
      "temperature": 0.15,
      "timeoutMs": 8000
    }
  }
}
```

Whatever goes into the model should be operational evidence, not raw private content. Redact message bodies, SMTP credentials, DKIM private keys, webhook secrets, API tokens, and license keys. The model can explain anomalies and draft recommendations, but any actual traffic change still needs approval mode, guarded SSL policy, or explicit operator action.

24. Security and identity

The management console runs on authenticated users, OTP, recovery codes, durable sessions, login throttling, RBAC roles, tenant/stream scope, API tokens, and audit logs. CTO, Engineer, and Deliverability modes just change what view you're working in — they don't change the underlying authentication requirement. A scoped operator can only inspect or operate on the tenants and streams they're allowed to; anything out of scope returns a 404 on that message id and logs a denied audit event.

25. Secret management

Secrets get referenced, never pasted straight into config. Supported providers include environment variables, local root-readable files, AWS Secrets Manager, Azure Key Vault, GCP Secret Manager, HashiCorp Vault, and Kubernetes Secrets. Secret checks catch placeholder values, expired rotations, weak storage, missing DKIM/TLS material, and unsafe exposure in support bundles.

26. Durable runtime state and databases

Single-node installs can rely on local JSON durable state for users, sessions, config revisions, audit indexes, license lease state, and warmup state. Zeta clusters can use Postgres or Supabase for shared management state instead. The active mail spool always stays local to each delivery node — the database never substitutes for the queue filesystem on the hot path.

27. Licensing and runtime leases

Customer nodes apply a SignalMTA license key, activate the node, and keep a short-lived lease alive. License state governs edition, feature flags, instance count, expiry warnings, and edition caps. License updates hot-reload entitlement state wherever that's possible. The public product only exposes customer-facing license application and status; issuing licenses on the company side stays separate.

28. Cluster routing

Zeta clustering centralizes visibility and policy while keeping delivery itself local. Routing decisions can weigh node heartbeat, worker load, queue depth, queue age, recipient-domain/MX remote acceptance, deferral rate, source-IP availability, pool standing, warmup state, stream stickiness, priority, TLS/DKIM identity, and policy compatibility. High-priority traffic only routes to nodes that can actually meet its route and identity requirements.

29. Kubernetes operation

Kubernetes deployments use StatefulSets for delivery nodes, persistent volumes for spool paths, ConfigMaps for non-secret config, Secrets for credentials, Services for API/console exposure, and probes covering spool, DNS, license, queue, and API health. Base autoscaling decisions on queue age, provider capacity, source-IP capacity, and export backlog rather than CPU alone.

30. Docker and Compose

Docker works well for evaluation, packaging validation, and controlled installs. Production containers need persistent volumes for spool, logs, runtime state, and configuration — never run a production container with ephemeral spool storage. Compose can bring up the MTA, console, Prometheus, and Grafana stack together for local qualification.

31. Outbound proxy and source-IP control

Outbound proxy mode is for operators who'd rather not bind every sending IP directly to the MTA host's operating system. SignalMTA still makes the delivery decision, but the network layer owns the final source address. That's handy for service providers, Kubernetes operators, leased IPv4 users, dedicated-IP customers, and anyone who keeps application servers separate from egress infrastructure.

Direct source-IP binding — Single host or simple bare-metal deployment. — Delivery worker binds directly to the configured local IP.

HAProxy TCP egress — Centralized source-IP control with familiar network tooling. — SignalMTA selects proxy target and records intended source-IP identity.

Custom SMTP-aware proxy — Service providers that need policy, tenant, and source-IP controls in the proxy layer. — SignalMTA passes route intent, tenant, pool, trace id, and delivery metadata to the proxy.

Linux policy routing or namespaces — Dedicated routing tables per pool or per tenant. — SignalMTA maps pools to routing marks or namespace targets.

Kubernetes egress gateway — Containerized Zeta deployments where pods should not own public IPs. — SignalMTA routes through approved egress gateways and logs gateway/source-IP attribution.

```
{
  "outboundProxy": {
    "mode": "enabled",
    "defaultPolicy": "require-approved-proxy",
    "proxies": [{
      "id": "egress-east-a",
      "type": "haproxy-tcp",
      "endpoint": "10.40.12.20:2525",
      "healthCheck": "smtp-banner",
      "allowedPools": ["transactional-east", "marketing-east"],
      "sourceIpMap": {
        "203.0.113.44": "proxy-slot-44",
        "203.0.113.45": "proxy-slot-45"
      }
    }]
  },
```

```
  "ipPools": [{
    "id": "transactional-east",
    "proxy": "egress-east-a",
    "sourceIps": ["203.0.113.44", "203.0.113.45"],
    "assignment": "dedicated-or-shared-by-policy"
  }]
}
```

For HAProxy-style TCP egress, keep your health checks strict. If a proxy is reachable but not sending from the expected source IP, treat it as unhealthy for that pool. Use route validation to confirm reverse DNS, SPF includes, DKIM alignment, TLS policy, warmup state, and provider caps before you let traffic through a proxy-owned IP.

```
# WebUI workflow
```

- In Configuration, go to IP Pools, click New pool, and set Source IP ownership to Proxy managed.
- In Configuration, go to Outbound Proxies, click Add proxy, and map the source IP slots.
- In Traffic Shaping, click Explain to confirm the pool/domain/source-IP cap.
- In Logs, open Trace and check mta_node, proxy_id, intended_source_ip, and observed_source_ip.

```
# CLI/API equivalents
./bin/signalmta proxy validate --proxy egress-east-a --pool transactional-east
./bin/signalmta routes explain --stream transactional --domain gmail.com --show-proxy
GET /api/proxies/egress-east-a/health
GET /api/logs/trace?trace_id=trc_01J1ABCD
```

Logs and webhooks carry mta_node, proxy_id, pool, intended_source_ip, observed_source_ip, egress_region, and route_id. If the proxy's observed IP doesn't match what SignalMTA intended, the attempt gets flagged source_ip_mismatch, SignalAI raises an advisory, and guarded policies can pause that pool before the reputation damage spreads.

32. IPv4 leasing integration

IPXO support is an inventory and readiness integration for leased IPv4 space, including the leased /24 blocks service providers and higher-volume senders commonly use. SignalMTA doesn't treat freshly leased IPs as instantly production-ready. It imports approved ranges, tracks assignment metadata, checks DNS and compliance prerequisites, connects the range to pools or proxies, and holds the IPs in warmup until operator policy clears them for normal traffic.

Import — Lease id, CIDR/range such as 203.0.113.0/24, status, renewal date, abuse contact, region, customer tag. — Inventory appears as leased, inactive IP space.

Ownership and compliance — Required approval state, allowed use, internal tenant assignment, abuse desk contact, renewal risk. — Unapproved ranges cannot be routed.

DNS readiness — Reverse DNS, forward-confirmed hostname, SPF include guidance, return-path alignment, TLS name planning. — IP stays blocked until identity checks pass.

Pool/proxy mapping — Pool assignment, dedicated/shared policy, proxy slot, region, failover target. — IP can be selected only by compatible routes.

Warmup — Daily cap, recipient groups, complaint rate, hard-bounce rate, deferrals, provider-specific limits. — Volume ramps gradually and pauses on risk signals.

Renewal/retirement — Lease expiry, renewal state, active queue use, open retries, reporting retention. — Operators drain traffic before returning or replacing IPs.

```
{
  "ipxo": {
    "enabled": true,
    "apiBase": "https://api.ipxo.com",
    "tokenRef": "env://IPXO_API_TOKEN",
    "syncIntervalMinutes": 30,
    "defaultImportState": "inventory_only",
    "requireOperatorApproval": true,
    "renewalWarningDays": 30
  },
  "leasedIpPolicy": {
    "requireReverseDns": true,
    "requireForwardConfirmedRdns": true,
    "requireWarmupPlan": true,
    "requireProxySlotValidation": true,
    "blockIfLeaseExpiresWithinDays": 7,
    "defaultBlockSize": "/24",
    "splitIntoPools": "operator-approved"
  }
}
```

The WebUI path is Configuration -> IP Inventory -> IPXO. Operators connect the API token by secret reference, sync inventory, review the imported ranges, assign approved IPs to a pool or a dedicated customer, attach warmup, and publish a config revision. For /24 blocks, don't dump all 256 addresses into one undifferentiated production pool — split the block into operator-approved sub-pools by use case, tenant, warmup phase, proxy slot, region, or provider strategy so reporting and shaping stay precise.

```
./bin/signalmta ipxo sync --dry-run
./bin/signalmta ipxo inventory --status pending-approval
./bin/signalmta ipxo approve --range 203.0.113.0/24 --tenant acme --pool inventory-only
./bin/signalmta ipxo split --range 203.0.113.0/24 --into warmup-a=/28,transactional-a=/28,marketing-a=/27
./bin/signalmta ip validate --ip 203.0.113.44 --checks rdns,spf,proxy,warmup
POST /api/integrations/ipxo/sync
POST /api/ip-inventory/203.0.113.44/approve
```

Attempt logs and reports keep the lease context around, so you can tell which customer, stream, proxy, pool, and leased range were involved in a delivery problem. Typical exported fields include lease_provider, lease_id, leased_range, leased_block, lease_expires_at, source_ip_owner, pool, proxy_id, warmup_phase, and ip_tenancy.

33. IP warmup automation

SignalWarmup is the automated warmup controller. It stages cold IPs with start dates, initial caps, target caps, overflow pools, recipient groupings, pause/resume state, and guardrail thresholds. In automatic mode it evaluates acceptance, complaint rate, hard-bounce rate, provider deferrals, block-oriented replies, reverse DNS, DKIM/SPF alignment, how TLS is set up, and eligible queue

depth, then decides to increase, wait, freeze, or pause based on policy. A troubled grouping can pause on its own without pausing the whole IP, and traffic over the active cap can spill into a warm overflow pool.

```
GET /api/ip-warmup
POST /api/ip-warmup/plan
POST /api/ip-warmup/evaluate
POST /api/ip-warmup/pause
POST /api/ip-warmup/resume
```

The WebUI path is Configuration -> Warmup. Use it to build automated plans, check what the controller will do next, review cap history, see overflow routing, pause or resume a source IP or provider grouping, and export warmup events for deliverability review.

34. Migration from PowerMTA and Postfix

The migration API reads legacy configuration and turns it into a SignalMTA config patch plus a report. PowerMTA virtual MTAs map to pools, source IPs map to pool source IPs, domain max rates map to throttle rules, and use-virtual-mta maps to submission-flow routes. Postfix mynetworks maps to allowed relay CIDRs, message_size_limit maps to message size, and relayhost maps to relay provider routes. Secrets get converted to secret references — they're never copied over as plaintext.

```
POST /api/config/migrate
{
  "type": "powermta",
  "text": "<virtual-mta vmta-a>..."
}
```

35. External delivery service connectors

SignalMTA can route selected traffic through providers such as SendGrid, Mailgun, SparkPost, or other SMTP/API services while the rest of your traffic stays on native delivery. Connector routes carry their own credentials, rate caps, retry behavior, bounce/complaint ingestion, route labels, logs, and cost reporting. Keep provider-routed traffic tagged separately so its reputation and cost don't get mixed into native delivery metrics.

36. Performance qualification

Performance qualification moves through stages: queue admission, smart-sink delivery, DNS pressure, TLS negotiation, provider simulation, restart and disaster recovery, export overhead, and finally controlled live sending. Every stage leaves behind evidence you can compare future changes against as a baseline. Edition caps are product boundaries, not a universal promise about throughput.

37. Support bundles

Support bundles gather diagnostics, config validation, production safety checks, a queue summary, logs, version info, OS facts, license state, recent events, and go-live readiness. They redact passwords, bearer tokens, API keys, DKIM private keys, webhook secrets, license signing material,

and message bodies.

38. Update lifecycle

Update checks read the release manifest, current version, license state, queue thresholds, runtime state, and the rollback path. The resulting update plan reports blockers, drain requirements, restart impact, backup plan, and verification steps. Don't restart a live delivery node while queue depth or active deliveries are above the plan's thresholds.

39. Production safety gate

The production safety gate keeps public SMTP shut off until explicit mode, an active license, relay authentication, inbound TLS, DKIM, bounce processing, logs, durable paths, DNS cache, TLS verification, rate limits, and basic warmup setup all check out. Warnings flag things like missing webhooks or missing IP warmup — gaps worth fixing before you push real volume through.

40. CLI and API surface

The CLI is built for the operator work you'll repeat over and over: bootstrap admin, apply license, validate config, publish config, inspect queues, repair spool, test domain auth, run diagnostics, qualify throughput, export support bundles, check updates, and run go-live audits. The API exposes that same operational model for automation and for the WebUI itself.

Commands follow one consistent shape: ./bin/signalmta <command> <subcommand> --config /etc/signalmta/signalmta.json --json. Commands that just inspect state are read-only by default. Commands that publish configuration, change queues, activate licenses, or modify users need an explicit actor/audit reason in production profiles.

bootstrap-admin — Creates the first console administrator, OTP configuration, and recovery workflow. — First-run setup.

license apply/status/activate — Applies a customer license key, reloads entitlement state, activates the node lease, and reports edition/node status. — Settings -> License.

evaluation status — Shows Guided setup, Expert mode, or Import existing config recommendations, severe blockers, and qualification configuration. — Evaluation.

config validate/diff/publish/rollback — Validates candidate config, shows revision diffs, publishes safe changes, or restores a prior revision. — Settings -> Configuration revisions.

domain-auth-plan — Generates DKIM, SPF/DMARC guidance, return-path records, DNS checks, and domain verification steps. — Domains -> Add domain.

queue summary/inspect/hold/release — Inspects queue state and controls held/deferred traffic with audit reasons. — Queues.

spool --check --repair-safe — Checks spool integrity, stale temporary files, metadata, and safe repair operations. — Queues -> Integrity.

shaping explain — Explains active recipient-domain/MX/pool/source-IP caps, response-driven triggers, and retry decisions. — Traffic Shaping -> Explain policy.

bounces classify — Classifies SMTP replies or DSNs and shows evidence, category, and default action. — Deliverability -> Bounces -> Explain.

ipxo sync/inventory/approve/split — Imports leased IPv4 ranges, approves inventory, and splits leased blocks into operating pools. — Configuration -> IP Inventory -> IPXO.

proxy validate — Validates HAProxy/custom egress health, source-IP mapping, pool compatibility, and observed source IP. — Configuration -> Outbound Proxies.

logs tail/export/trace — Reads delivery-attempt, SMTP connection, webhook, and trace logs with filtering and redaction. — Observability -> Logs.

performance-qualification, qualify-host — Runs capacity and readiness checks for the customer environment. — Readiness -> Performance qualification.

support-bundle — Builds a redacted diagnostic bundle for support or internal incident review. — Support -> Export bundle.

node drain/activate/status — Controls HA node maintenance and cluster routing state. — Cluster -> Nodes.

```
# First administrator and license
./bin/signalmta bootstrap-admin --config /etc/signalmta/signalmta.json --email ops@example.com --require-
./bin/signalmta license apply --key-file ./license.key --state /var/lib/signalmta/runtime-license.json
./bin/signalmta license status --json

# Configuration lifecycle
./bin/signalmta config validate --config ./candidate.json
./bin/signalmta config diff --from active --to ./candidate.json
./bin/signalmta config publish --config ./candidate.json --reason "Add Gmail warmup pool" --actor ops@exa
./bin/signalmta config rollback --to rev_20260714_001 --reason "revert provider cap"

# Delivery operations
./bin/signalmta queue summary --group-by priority,domain,pool,state
./bin/signalmta shaping explain --domain gmail.com --pool warmup-a --source-ip 203.0.113.44
./bin/signalmta logs trace --message-id msg_01J1EFGH
./bin/signalmta bounces classify --sample ./dsn.eml --explain

# IP inventory and egress
./bin/signalmta ipxo sync --dry-run
./bin/signalmta ipxo split --range 203.0.113.0/24 --into warmup-a=/28,marketing-a=/27
./bin/signalmta proxy validate --proxy egress-east-a --pool marketing-a

# Readiness and support
./bin/signalmta qualify-host --profile alpha-250k --target-mph 250000 --report-dir /var/log/signalmta/rea
./bin/signalmta support-bundle --since 24h --redact --output ./support-bundle.tgz
```

Reach for the CLI when you need repeatable automation, CI/CD config promotion, scripted diagnostics, or terminal-based work. Reach for the WebUI when you want guided forms, inline validation, visual diffs, role-based workflows, and easier troubleshooting. Either way, both paths write to the same audit log and drive the same runtime behavior.

Troubleshooting Runbooks

When something's wrong, start by narrowing down the scope as far as it'll go: tenant, stream, campaign, tag, domain, MX, pool, source IP, route, or error category. Don't jump to treating every delay as if the whole MTA is down.

Queue age rising — Priority lane, recipient-domain/MX limits, worker count, spool watermark. — Protect high priority, identify throttled destinations, pause noisy streams.

Gmail or major domain slowing — MX deferrals, pool reputation, retry curve, connection cap. — Lower domain rate, spread across approved pools, preserve provider evidence.

Block-oriented bounces — Source IP, domain, stream, raw reply, provider profile. — Hold affected scope and route to deliverability review.

Authentication failures — DKIM selector, DNS propagation, SPF/DMARC alignment, return-path. — Stop scale-up until records pass from multiple resolvers.

TLS failures — TLS mode, remote MX support, certificate verification, SNI. — Fix route policy or isolate the destination; do not retry aggressively.

Webhook backlog — Target latency, HTTP status, signing failures, retry queue. — Increase export workers or pause noncritical targets.

```
./bin/signalmta diagnostics domain --domain gmail.com --since 2h
./bin/signalmta diagnostics stream --stream transactional --include-headers
./bin/signalmta diagnostics tls --mx mx.example.net
./bin/signalmta support-bundle --since 24h --redact
```

SignalMTA SignalAI screenshot

SignalAI: catches anomalies, gives you explainable recommendations, and routes them through an operator approval flow.

Functional Area Reference

This table maps out SignalMTA's major functional areas along with the tuning options operators reach for most.

SMTP receiving — listen ports, TLS, max size, relay sources, AUTH, mTLS, rate limits. — Security, Settings, Streams.

API injection — tokens, tenant scope, stream scope, idempotency, payload limits. — Settings, Security, Streams.

Queues — priority reserve, spool path, watermark, retry lanes, permanent failure policy. — Queues.

Delivery workers — worker count, connection caps, session limits, timeout policy. — Dashboard, Queues, Domains.

DNS — resolver list, cache TTLs, negative TTLs, concurrency, timeout. — Engineer diagnostics.

TLS — opportunistic/required, SNI, minimum version, certificate validation, ACME. — Security, Domains.

DKIM/DKIM2 — selectors, key refs, double signing, rotation, signed headers. — Domains.

Traffic shaping — recipient-domain/MX caps, stream caps, pool caps, response-driven backoff, cooldown. — Destinations, Streams, SignalAI.

Warmup — daily cap, growth rate, overflow pool, pause thresholds, provider grouping. — Deliverability.

Bounces/FBL — VERP, DSN parser, ARF processor, Feedback-ID correlation, redacted-recipient policy, categories, suppressions, exports. — Reports, Deliverability, Logs.

Exports — webhooks, JSONL, syslog, Kafka/AMQP, signing, retry, batch size. — Reports, Settings.

Cluster — node membership, policy distribution, shared database, route intelligence. — Fleet, Dashboard.

Security — OTP, RBAC, token scope, audit logs, secret refs, relay controls. — Security.

SignalAI — advisory thresholds, approval policy, anomaly scope, recommendation mode. — SignalAI.