

Perfuse Reference Manual

A healthcare integration engine: channels, formats, delivery and operation

version 0.1 · 64dcb01

Table of Contents

- 1 Introduction**
 - 1.1 What Perfuse is for
 - 1.2 What it replaces
 - 1.3 How to read this manual
 - 1.4 Conventions
- 2 Getting Started**
 - 2.1 Running it
 - 2.2 The first channel
 - 2.3 Building a channel from the interface
 - 2.4 Verifying it works before connecting anything
 - 2.5 What to do next
- 3 The Web Interface**
 - 3.1 Reading the header
 - 3.2 Monitor — whether it is working, and what happened
 - 3.2.1 Dashboard
 - 3.2.2 Messages
 - 3.2.3 Queue
 - 3.2.4 Alerts
 - 3.2.5 Metrics
 - 3.2.6 Flow map
 - 3.2.7 Sharing what a feed actually contains
 - 3.3 Build — change what a channel does, and try it before it is live
 - 3.3.1 Channels
 - 3.3.2 Scripts
 - 3.3.3 Contracts
 - 3.3.4 Tables
 - 3.3.5 Mapper
 - 3.3.6 Playground
 - 3.3.7 Shadow
 - 3.4 Exchange — other formats and other organisations
 - 3.4.1 Team dashboards
 - 3.4.2 FHIR lab

3.4.3 Subscriptions

3.4.4 Documents

3.4.5 Claims & auth

3.4.6 TEFCA

3.5 Administer — who can do what, and how this server is set up

3.5.1 Users

3.5.2 Certificates

3.5.3 Activity

3.5.4 Fleet

3.5.5 Migrate

3.5.6 Settings

3.6 What the colours mean

4 Channels

4.1 Anatomy

4.2 One source, many destinations

4.3 Destinations are not a chain

4.4 Naming

4.5 Enabling and disabling

4.6 Where channels live

4.7 Encrypting a channel's own listener

4.8 Validation

5 Message Flow

5.1 The order of events

5.2 When the acknowledgement is sent

5.3 When to choose on_receipt

5.4 Where the message is recorded

5.5 Failure at each stage

5.6 Ordering

5.7 Concurrency

6 Transformation

6.1 The steps

6.2 Clear, remove and the difference

6.3 Conditions

6.4 Paths

6.5 Errors

6.6 Order matters, and the trace shows it

6.7 Suggesting mappings, and what to do with a suggestion

6.7.1 Give it something to be confident about

6.7.2 Approval is one mapping at a time

6.7.3 Two things to do with what you approved

6.8 Why not a script

- 6.9 What a script is allowed to do
- 6.10 Handling a date that does not match its format
- 6.11 The scripting languages
 - 6.11.1 What the output means

7 Filtering

- 7.1 Filtered is not failed
- 7.2 What a filter can read
- 7.3 Expressions
- 7.4 An error in the filter is a refusal
- 7.5 Filter at the source where you can
- 7.6 Testing a filter

8 Message Formats

- 8.1 HL7 v2
- 8.2 HL7 v3 and CDA
- 8.3 X12
 - 8.3.1 Splitting an interchange
 - 8.3.2 When the counts do not add up
 - 8.3.3 Transactions Perfuse reads as structured data
 - 8.3.4 Eligibility, and the CAQH CORE data content rule
 - 8.3.5 Prior authorisation
 - 8.3.6 Attachments, and why they need special handling
 - 8.3.7 Electronic signatures on attachments
 - 8.3.8 Binary segments
- 8.4 FHIR
 - 8.4.1 What an HL7 v2 message becomes
 - 8.4.2 US Core 9.0.0 and USCDI v6
 - 8.4.3 R4 in both directions
 - 8.4.4 Subscriptions
- 8.5 NCPDP Telecommunication Standard
- 8.6 NCPDP SCRIPT
- 8.7 DICOM
 - 8.7.1 What a C-STORE listener accepts
- 8.8 Delimited
 - 8.8.1 Naming the columns
 - 8.8.2 What a filter or a script sees
 - 8.8.3 The three that change what counts as an error
- 8.9 Raw
- 8.10 Choosing
- 8.11 SMART Health Links and Cards

9 Acknowledgements

- 9.1 Timing

- 9.2 What Perfuse sends
- 9.3 Identifying this engine
- 9.4 The trigger event
- 9.5 Negative acknowledgements are not failures to hide
- 9.6 Timeouts
- 9.7 Sources that do not acknowledge
- 9.8 X12 acknowledgements
- 9.9 HL7 v3 acknowledgements

10 Delivery, Queueing and Retry

- 10.1 Queue per destination
- 10.2 Queue settings
- 10.3 Ordering is preserved
- 10.4 When queueing is disabled
- 10.5 Retry versus queue
- 10.6 What an HTTP endpoint counts as delivered
- 10.7 Naming and pruning an archive folder
- 10.8 Sending to a FHIR server
 - 10.8.1 Hosted FHIR services
 - 10.8.2 Public health case reporting
 - 10.8.3 Lab results to public health: ELR
- 10.9 When the transport says yes and the message did not arrive
- 10.10 What a delivery outcome means
- 10.11 Slow delivery
- 10.12 Writing to a database
- 10.13 Sending to a raw socket
- 10.14 Amazon S3, SQS and SNS
- 10.15 AMQP 1.0 and Azure
- 10.16 MQTT
- 10.17 IBM MQ
- 10.18 Google Cloud Pub/Sub and Cloud Storage
- 10.19 Delivery to another channel
- 10.20 What happens when the process is killed
 - 10.20.1 Acknowledging on delivery
 - 10.20.2 Acknowledging on receipt
 - 10.20.3 A sender that dies half way through a message
 - 10.20.4 A full disk
 - 10.20.5 What was not tested

11 Alerting

- 11.1 The alert kinds
- 11.2 Silence is the hardest thing to detect
- 11.3 Rhythm-aware detection

- 11.3.1 Median, not mean
- 11.3.2 The floor is a percentile, not the minimum
- 11.3.3 It will say when it cannot tell

- 11.4 Setting alerts up
- 11.5 Where alerts go

12 The Message Store

- 12.1 What is kept
- 12.2 Searching
- 12.3 Replay and reprocessing
- 12.4 Staleness
- 12.5 Retention
- 12.6 The store holds clinical data

13 Debugging

- 13.1 Tracing a message
- 13.2 Step by step
- 13.3 Four outcomes, and the distinction that matters
- 13.4 Testing a change before saving it
- 13.5 What tracing cannot tell you
- 13.6 Scripts

14 Testing

- 14.1 Generated test traffic
 - 14.1.1 Why it contains no real data
- 14.2 Building a channel from a sample
 - 14.2.1 Three lists
 - 14.2.2 The separator warning
- 14.3 Comparing against the old engine
- 14.4 Verifying against the real thing, in containers
- 14.5 Diagnosing a test that fails once and never again
- 14.6 What still needs real infrastructure
- 14.7 Measured against public conformance corpora
 - 14.7.1 FHIR R4, the specification's own examples
 - 14.7.2 HL7 v2, the HAPI test corpus
 - 14.7.3 What has not been run

15 Migration from Mirth

- 15.1 Importing channels
- 15.2 What translates and what needs a decision
- 15.3 What the importer has been tested against
- 15.4 Proving it matches
 - 15.4.1 Four numbers, not a pass rate
 - 15.4.2 Differences are grouped by field
 - 15.4.3 The verdict does not round

- 15.5 A suggested sequence
- 15.6 What only runs on one vendor's software
 - 15.6.1 The distinction is the whole point
 - 15.6.2 External scripts are reported as unknown
 - 15.6.3 Perfuse's own lock-in, stated in the same breath
 - 15.6.4 Setting up step 6
- 15.7 Going back the other way

16 CMS-0057 for payers

- 16.1 The four APIs
- 16.2 Coverage requirements and documentation: CRD and DTR
- 16.3 Claims as CARIN Blue Button
- 16.4 Prior authorisations as PDex
- 16.5 The Provider Access and Payer-to-Payer APIs
- 16.6 Prior authorisation metrics
- 16.7 What was checked, and against what
- 16.8 Deciding prior authorisation requests: PAS
 - 16.8.1 The reviewer queue
 - 16.8.2 Receiving the documents a pended request asks for
 - 16.8.3 Deciding in an X12 utilization management system

17 Security

- 17.1 Transport security
- 17.2 Authentication and roles
- 17.3 Who may call the FHIR endpoint
- 17.4 Audit
- 17.5 Secrets in configuration
- 17.6 What is in the database
- 17.7 Tenants
- 17.8 Test data
- 17.9 Single sign-on
 - 17.9.1 What a group mapping decides
 - 17.9.2 SAML, specifically
 - 17.9.3 What each provider sends
- 17.10 What Perfuse does not do

18 Operations

- 18.1 Deployment
- 18.2 Configuration as files
- 18.3 Backup
- 18.4 Restart behaviour
- 18.5 Upgrading
- 18.6 Monitoring
- 18.7 Capacity

- 18.8 Where this installation got stuck
- 18.9 Logs
- 18.10 Running two instances
- 18.11 More than one server
 - 18.11.1 Adding a peer
 - 18.11.2 Controlling a peer, which is off
 - 18.11.3 What the report says
 - 18.11.4 Clock skew
 - 18.11.5 What a peer learns about you
 - 18.11.6 A peer pointed at itself
 - 18.11.7 TLS on internal networks

19 The Command Line

- 19.1 Running channels
 - 19.1.1 perfuse serve
 - 19.1.2 perfuse run
 - 19.1.3 perfuse check
 - 19.1.4 perfuse test
- 19.2 Migrating from another engine
 - 19.2.1 perfuse explain
 - 19.2.2 perfuse translate
 - 19.2.3 perfuse compare
- 19.3 Understanding a feed
 - 19.3.1 perfuse profile
 - 19.3.2 perfuse contract
- 19.4 Producing messages
 - 19.4.1 perfuse generate
 - 19.4.2 perfuse deident
- 19.5 Moving messages by hand
 - 19.5.1 perfuse listen
 - 19.5.2 perfuse send
- 19.6 FHIR
 - 19.6.1 perfuse fhir
 - 19.6.2 FHIR flags on perfuse serve
- 19.7 Setting up and operating
 - 19.7.1 perfuse init
 - 19.7.2 perfuse service
 - 19.7.3 perfuse token
 - 19.7.4 perfuse athena
 - 19.7.5 perfuse sbom
 - 19.7.6 perfuse version

20 Channel Reference

20.1 Channel

- 20.1.1 name
- 20.1.2 group
- 20.1.3 attachments
- 20.1.4 delimited
- 20.1.5 tables
- 20.1.6 enabled
- 20.1.7 dataType
- 20.1.8 x12
- 20.1.9 script
- 20.1.10 dicom
- 20.1.11 filter
- 20.1.12 transformations
- 20.1.13 scripts
- 20.1.14 shadow

21 Source Reference

21.1 AMQP Source

- 21.1.1 prefetch
- 21.1.2 reconnect

21.2 Azure Blob Source

- 21.2.1 move_to
- 21.2.2 error_prefix
- 21.2.3 poll_interval
- 21.2.4 max_object_size

21.3 Broker Source

- 21.3.1 addr
- 21.3.2 destination
- 21.3.3 login
- 21.3.4 passcode
- 21.3.5 host
- 21.3.6 selector
- 21.3.7 subscription_id
- 21.3.8 heartbeat
- 21.3.9 reconnect
- 21.3.10 timeout

21.4 DICOM Query Source

- 21.4.1 address
- 21.4.2 called_ae
- 21.4.3 calling_ae
- 21.4.4 level
- 21.4.5 match

- 21.4.6 return
- 21.4.7 interval
- 21.4.8 window
- 21.4.9 overlap
- 21.4.10 limit
- 21.4.11 timeout
- 21.4.12 emit_on_first_poll
- 21.5 DICOM Source
 - 21.5.1 listen
 - 21.5.2 ae_title
 - 21.5.3 allowed_calling_ae
 - 21.5.4 sop_classes
 - 21.5.5 transfer_syntaxes
 - 21.5.6 max_object_bytes
 - 21.5.7 idle_timeout
 - 21.5.8 tls
- 21.6 Database Source
 - 21.6.1 driver
 - 21.6.2 dsn
 - 21.6.3 query
 - 21.6.4 after_query
 - 21.6.5 key_column
 - 21.6.6 template
 - 21.6.7 batch_size
 - 21.6.8 max_attempts
 - 21.6.9 query_timeout
- 21.7 FTP Source
 - 21.7.1 host
 - 21.7.2 user
 - 21.7.3 password
 - 21.7.4 security
 - 21.7.5 root
 - 21.7.6 timeout
- 21.8 File Source
 - 21.8.1 root
- 21.9 GCS Source
 - 21.9.1 move_to
 - 21.9.2 error_prefix
 - 21.9.3 poll_interval
 - 21.9.4 max_object_size
- 21.10 HTTP Source

- 21.10.1 listen
- 21.10.2 path
- 21.10.3 token
- 21.10.4 max_message_size
- 21.11 IBMMQ Source
 - 21.11.1 poll_interval
- 21.12 Java Script Source
 - 21.12.1 poll_interval
 - 21.12.2 timeout
- 21.13 Kafka Source
 - 21.13.1 brokers
 - 21.13.2 topics
 - 21.13.3 group
 - 21.13.4 from_beginning
 - 21.13.5 commit_after_delivery
 - 21.13.6 max_message_size
 - 21.13.7 session_timeout
 - 21.13.8 timeout
- 21.14 MQTT Source
 - 21.14.1 reconnect
- 21.15 Pub Sub Source
 - 21.15.1 max_messages
 - 21.15.2 reconnect
- 21.16 S3 Source
 - 21.16.1 prefix
 - 21.16.2 after_read
 - 21.16.3 move_to
 - 21.16.4 error_prefix
 - 21.16.5 poll_interval
 - 21.16.6 max_object_size
 - 21.16.7 framed
- 21.17 SFTP Source
 - 21.17.1 host
 - 21.17.2 password
 - 21.17.3 key_file
 - 21.17.4 known_hosts_file
 - 21.17.5 insecure_skip_host_key_check
 - 21.17.6 pattern
 - 21.17.7 move_to
 - 21.17.8 stable_for
 - 21.17.9 framed

- 21.17.10 max_file_size
- 21.17.11 timeout
- 21.18 SMB Source
 - 21.18.1 host
 - 21.18.2 share
 - 21.18.3 user
 - 21.18.4 password
 - 21.18.5 domain
 - 21.18.6 timeout
- 21.19 SOAP Source
 - 21.19.1 listen
 - 21.19.2 path
 - 21.19.3 version
 - 21.19.4 element
 - 21.19.5 base64
 - 21.19.6 response_element
 - 21.19.7 wsdl
 - 21.19.8 username
 - 21.19.9 password
 - 21.19.10 max_message_size
 - 21.19.11 fault_on_nak
- 21.20 SQS Source
 - 21.20.1 visibility_timeout
- 21.21 Serial Source
 - 21.21.1 baud
 - 21.21.2 parity
 - 21.21.3 stop_bits
 - 21.21.4 flow_control
 - 21.21.5 quiet_after
 - 21.21.6 reopen_after
- 21.22 Source
 - 21.22.1 type
 - 21.22.2 max_message_size
 - 21.22.3 idle_timeout
 - 21.22.4 max_connections
- 21.23 TCP Source
 - 21.23.1 idle_timeout
 - 21.23.2 max_connections
 - 21.23.3 reply
- 21.24 Web DAV Source
 - 21.24.1 user

21.24.2 password

21.24.3 timeout

22 Destination Reference

22.1 AMQP Destination

22.1.1 content_type

22.2 Azure Blob Destination

22.2.1 blob

22.2.2 tier

22.3 Broker Destination

22.3.1 addr

22.3.2 destination

22.3.3 login

22.3.4 passcode

22.3.5 host

22.3.6 content_type

22.3.7 headers

22.3.8 persistent

22.4 CDA Destination

22.4.1 url

22.4.2 dir

22.4.3 write

22.4.4 identifier_systems

22.4.5 require_agreement

22.4.6 on_no_document

22.4.7 bearer_token

22.5 Channel Destination

22.6 DICOM Destination

22.6.1 address

22.6.2 called_ae

22.6.3 calling_ae

22.6.4 transfer_syntaxes

22.6.5 timeout

22.7 Database Destination

22.7.1 driver

22.7.2 statement

22.7.3 params

22.7.4 timeout

22.7.5 max_open_conns

22.8 Destination

22.8.1 name

22.8.2 elr

- 22.8.3 response_transformer
- 22.8.4 queue
- 22.9 Document Destination
 - 22.9.1 format
 - 22.9.2 template
 - 22.9.3 font_size
 - 22.9.4 temp_suffix
 - 22.9.5 timeout
- 22.10 ECR Destination
 - 22.10.1 rtc
- 22.11 ELR Destination
 - 22.11.1 config
 - 22.11.2 rtc
- 22.12 FHIR Destination
 - 22.12.1 url
 - 22.12.2 version
 - 22.12.3 timezone
 - 22.12.4 default_identifier_system
 - 22.12.5 claim_us_core
 - 22.12.6 validate_before_send
 - 22.12.7 reject_on_warning
 - 22.12.8 bearer_token
 - 22.12.9 ecr
- 22.13 FTP Destination
 - 22.13.1 host
 - 22.13.2 user
 - 22.13.3 password
 - 22.13.4 security
 - 22.13.5 insecure_skip_verify
 - 22.13.6 allow_clear_password
 - 22.13.7 temp_suffix
 - 22.13.8 timeout
- 22.14 GCS Destination
 - 22.14.1 object
- 22.15 HTTP Destination
 - 22.15.1 url
 - 22.15.2 method
 - 22.15.3 content_type
 - 22.15.4 bearer_token
 - 22.15.5 success_status
 - 22.15.6 fail_on_body

- 22.15.7 follow_redirects
- 22.16 IBMMQ Destination
 - 22.16.1 non_persistent
- 22.17 Java Script Destination
 - 22.17.1 script
 - 22.17.2 timeout
 - 22.17.3 success_on_undefined
- 22.18 Kafka Destination
 - 22.18.1 brokers
 - 22.18.2 topic
 - 22.18.3 key
 - 22.18.4 acks
 - 22.18.5 compression
 - 22.18.6 headers
 - 22.18.7 timeout
- 22.19 MQTT Destination
 - 22.19.1 topic
- 22.20 Pub Sub Destination
- 22.21 S3 Destination
 - 22.21.1 key
 - 22.21.2 access_key_id
 - 22.21.3 secret_access_key
 - 22.21.4 path_style
 - 22.21.5 content_type
 - 22.21.6 timeout
 - 22.21.7 storage_class
 - 22.21.8 format
- 22.22 SFTP Destination
 - 22.22.1 host
 - 22.22.2 password
 - 22.22.3 file_name
 - 22.22.4 temp_suffix
 - 22.22.5 timeout
- 22.23 SMTP Destination
 - 22.23.1 host
 - 22.23.2 from
 - 22.23.3 to
 - 22.23.4 cc
 - 22.23.5 bcc
 - 22.23.6 body
 - 22.23.7 attach_name

- 22.23.8 username
- 22.23.9 password
- 22.23.10 starttls
- 22.23.11 insecure_skip_verify

22.24 SNS Destination

- 22.24.1 subject

22.25 SOAP Destination

- 22.25.1 action
- 22.25.2 body
- 22.25.3 header
- 22.25.4 fault_is_success
- 22.25.5 timeout

22.26 SQS Destination

- 22.26.1 queue_url

22.27 TCP Dest

- 22.27.1 expect_reply
- 22.27.2 keep_alive

23 Format Reference

23.1 Attachments

- 23.1.1 reassemble

23.2 Delimited

- 23.2.1 delimiter
- 23.2.2 quote
- 23.2.3 split
- 23.2.4 keep_blank_lines
- 23.2.5 filter
- 23.2.6 transformations

23.3 HL7v3 Options

- 23.3.1 filter
- 23.3.2 acknowledge
- 23.3.3 sender_device
- 23.3.4 transformations

23.4 TCP Framing

- 23.4.1 framing
- 23.4.2 start_block
- 23.4.3 keep_delimiter
- 23.4.4 trim_padding

23.5 X12 Options

- 23.5.1 envelope
- 23.5.2 split
- 23.5.3 acknowledge

- 23.5.4 ack_sender_id
- 23.5.5 ack_sender_qualifier
- 23.5.6 transformations

24 Behaviour Reference

- 24.1 AMQP Connection
 - 24.1.1 username
 - 24.1.2 password
 - 24.1.3 hostname
 - 24.1.4 tls
 - 24.1.5 timeout
- 24.2 AWS Access
 - 24.2.1 secret_access_key
 - 24.2.2 session_token
- 24.3 Ack
 - 24.3.1 include_trigger_event
- 24.4 Azure Blob Access
 - 24.4.1 key
 - 24.4.2 sas
- 24.5 Contract Ref
 - 24.5.1 check_every
 - 24.5.2 over
- 24.6 DICOM Options
- 24.7 ECR Facility
- 24.8 FHIR Auth
 - 24.8.1 region
 - 24.8.2 token_url
 - 24.8.3 client_id
 - 24.8.4 client_secret
 - 24.8.5 scope
- 24.9 File Poll
 - 24.9.1 pattern
 - 24.9.2 error_dir
 - 24.9.3 raw
 - 24.9.4 batch_size
 - 24.9.5 sort_by
- 24.10 GCP Access
 - 24.10.1 credentials_file
- 24.11 GCS Access
- 24.12 IBMMQ Connection
 - 24.12.1 username
 - 24.12.2 password

- 24.12.3 timeout
- 24.13 Kafka SASL
 - 24.13.1 mechanism
 - 24.13.2 password
- 24.14 MQTT Connection
 - 24.14.1 client_id
 - 24.14.2 username
 - 24.14.3 password
 - 24.14.4 timeout
- 24.15 NCPDP Options
 - 24.15.1 transformations
- 24.16 Queue Config
 - 24.16.1 max_attempts
 - 24.16.2 max_backoff
 - 24.16.3 max_depth
 - 24.16.4 retain_hours
- 24.17 Retry
 - 24.17.1 attempts
 - 24.17.2 backoff
- 24.18 Script Options
 - 24.18.1 transformations
- 24.19 Scripts
 - 24.19.1 filter
 - 24.19.2 transformer
 - 24.19.3 preprocessor
 - 24.19.4 deploy
 - 24.19.5 undeploy
 - 24.19.6 postprocessor
 - 24.19.7 language
 - 24.19.8 timeout
 - 24.19.9 include
 - 24.19.10 file_roots
 - 24.19.11 allow
- 24.20 Shadow
 - 24.20.1 channel
 - 24.20.2 sample
 - 24.20.3 compare
 - 24.20.4 ignore
 - 24.20.5 max_differences
 - 24.20.6 timeout

25 How Each Component Works

- 25.1 hl7
- 25.2 admit
- 25.3 alerts
- 25.4 amqp
- 25.5 analyze
- 25.6 api
- 25.7 attach
- 25.8 authlimit
- 25.9 awsmsg
- 25.10 awsv4
- 25.11 azblob
- 25.12 branding
- 25.13 cda
- 25.14 channeltest
- 25.15 cms0057
- 25.16 codeset
- 25.17 compliance
- 25.18 config
- 25.19 contract
- 25.20 crd
- 25.21 dashboards
- 25.22 dbtime
- 25.23 delimited
- 25.24 dicom
- 25.25 drug
- 25.26 dsdr
- 25.27 e4x
- 25.28 egress
- 25.29 engine
- 25.30 eprescribe
- 25.31 equiv
- 25.32 expr
- 25.33 fdlimit
- 25.34 fhir
- 25.35 fhirserver
- 25.36 framing
- 25.37 ftpconn
- 25.38 gcp
- 25.39 generate
- 25.40 hl7dict
- 25.41 hl7v3

- 25.42 hl7xml
- 25.43 ibmmq
- 25.44 identity
- 25.45 kafka
- 25.46 ldap
- 25.47 mapper
- 25.48 metrics
- 25.49 mirth
- 25.50 mirthlive
- 25.51 mqtt
- 25.52 msgstore
- 25.53 ncpdp
- 25.54 oidc
- 25.55 parity
- 25.56 pas278
- 25.57 pdf
- 25.58 priorauth
- 25.59 publichealth
- 25.60 queue
- 25.61 s3put
- 25.62 saml
- 25.63 scim
- 25.64 script
- 25.65 settings
- 25.66 sftpconn
- 25.67 shadow
- 25.68 shl
- 25.69 smartauth
- 25.70 soap
- 25.71 sqldb
- 25.72 sqlitedb
- 25.73 steps
- 25.74 stomp
- 25.75 store
- 25.76 tefca
- 25.77 tenant
- 25.78 tlsconf
- 25.79 tomirth
- 25.80 trace
- 25.81 transform
- 25.82 translate

25.83 udap

25.84 v2fhir

25.85 vcs

25.86 vfs

25.87 vpn

25.88 web

25.89 webauthn

25.90 winservice

25.91 x12

25.92 x12xml

25.93 xmldsig

25.94 xtree

26 Index of Configuration Keys

26.1 All keys

1 Introduction

Perfuse moves clinical messages between systems that were never designed to talk to each other. A laboratory sends results in HL7 v2; the practice management system expects them with different field positions, a different patient identifier and an acknowledgement within five seconds. Something has to sit in the middle, and that something is an integration engine.

This manual describes what Perfuse does, every option it accepts, and — where a decision is not obvious — why it works the way it does.

1.1 What Perfuse is for

A single interface between two clinical systems is commonly costed in the tens of thousands, with fifteen to twenty per cent of that again every year in maintenance. Much of that cost is not the work; it is that the work requires somebody who has done HL7 before, and most practices do not employ one.

Perfuse is built on the assumption that the person configuring it has a problem to solve and no prior HL7 experience. That assumption shows up in specific places:

- Everything is configurable from the web interface. A feature reachable only from a terminal or a hand-edited file is treated as a defect, not a design choice.
- Messages can be traced one transformation at a time, showing what each step read, what it changed, and what it skipped and why.
- Test traffic can be generated that matches the shape of a real feed without containing any of its data.
- A channel can be proposed from a single sample message, with an explicit statement of what the sample cannot tell you.
- Before replacing an existing engine, its output and Perfuse's can be compared message by message, with the differences grouped by field.

The chapter on testing and the one on migration exist because the answer to "why should I trust this" has to be evidence you gather yourself, not a claim made here.

1.2 What it replaces

Perfuse covers the ground held by Mirth Connect, and can import Mirth channel exports directly. The chapter on migration describes what translates cleanly, what needs a decision, and how to prove the result matches before cutting over.

The differences that matter in daily use are debugging and silence. In Mirth, seeing what a transformer did to a message generally means redeploying the channel with logging added. In Perfuse, any recorded message can be replayed through the current configuration and inspected step by step, without touching the running channel. And where a Mirth alert on "no traffic" is a fixed threshold that is useless

on any feed that is quiet at night, Perfuse learns what each feed's normal week looks like and alerts against that.

1.3 How to read this manual

The first chapters are conceptual and are worth reading in order: channels, message flow, then transformation. Everything after that can be read as needed.

The reference chapters near the end list every configuration key, its type, whether it is required and its default. Those chapters are generated from the source, so they cannot describe an option that does not exist or omit one that does. If a key appears in a configuration file and not in the index, it is not a Perfuse key.

1.4 Conventions

Configuration is YAML. Most keys are snake_case on the wire — `max_message_size`, `idle_timeout` — and this manual always gives the wire form, because that is what you type. There is one long-standing exception, `dataType` on a channel, which is camelCase; it is documented that way in the channel reference and accepted only in that spelling.

Field references in HL7 use the usual notation: `PID-5` is the fifth field of the PID segment, `PID-5.1` its first component, and `PID-3[2]` the second repetition of the third field. Where a path is written without a repetition it addresses the first.

Durations are written as a number and a unit — `30s`, `5m`, `2h`. A bare number is not a duration and will be refused rather than guessed at.

2 Getting Started

Perfuse is a single binary with an embedded web interface. There is no application server to install, no database to provision and no separate administration console.

2.1 Running it

```
perfuse serve -channels ./channels -db ./perfuse.db -addr 0.0.0.0:8443 \
  -tls-cert ./cert.pem -tls-key ./key.pem
```

`-channels` is a directory of YAML files, one per channel. `-db` is a SQLite file holding recorded messages, users and audit history; it is created if it does not exist.

On first start with no users configured, an administrator account is created and its password written to the log once. That password is not recoverable afterwards.

Running without `-tls-cert` and `-tls-key` serves plain HTTP. That is acceptable on a loopback address for a first look and is not acceptable anywhere else: the web interface carries credentials and message content, both of which are readable on the wire without TLS. See security.

2.2 The first channel

A channel needs a name, a source and at least one destination. The smallest useful one receives HL7 over MLLP and writes each message to a file:

```
name: first-feed
dataType: hl7
source:
  type: mllp
  listen: "0.0.0.0:6661"
destinations:
  - name: to-disk
    type: file
    file:
      root: /var/spool/perfuse
      dir: incoming
```

Save that as `channels/first-feed.yaml` and it is picked up without a restart.

Three things about that file are worth noticing, because they are the shape of every channel:

- The source block is flat for MLLP. `listen` sits directly on `source`, not inside a `source.mllp` block. Transports that need more than an address — HTTP, SOAP, DICOM — do have their own sub-block, and those are listed in the source reference.
- A file destination needs both `root` and `dir`. `root` is the boundary that paths are resolved against and cannot escape; `dir` is where messages land within it.
- There is no transformation and no filter. Both are optional, and a channel with neither simply passes messages through.

2.3 Building a channel from the interface

Everything above can be done from the web interface instead, and for a first channel that is the better route because the form will not let you produce a file that does not load.

The builder offers three ways in:

1. **From a message somebody sent you.** Paste the sample the lab or vendor emailed. Perfuse reads it and proposes a channel, along with three separate lists: what the sample shows, what has been guessed, and what no sample can tell you. This is described in testing.
2. **From something that already works.** A set of complete channels that need only their addresses filled in.
3. **From nothing.** A blank form. Available, but it is the slowest of the three and the easiest to get wrong.

2.4 Verifying it works before connecting anything

Two things can be done before the sending system exists.

Send a message by hand from the interface — the message sender accepts pasted text and reports what happened to it, including which destinations accepted it and what each transformation did.

Or generate test traffic. If the channel has handled any messages at all, Perfuse can learn their shape and produce more messages like them, in the same proportions, with none of the original content. See testing.

2.5 What to do next

- Channels explains what a channel is and how its parts fit together.
- Message flow follows one message from arrival to delivery, which is the fastest way to understand where to put a change.
- Alerting is worth setting up before a feed goes live rather than after the first incident.

3 The Web Interface

Perfuse is built on the assumption that everything is doable from the browser, and a feature that can only be reached by editing a file is treated as a defect. This chapter is a tour of what is behind each part of the navigation, organised the way the interface is: by the question you are asking.

The bar carries **Dashboard** and **Channels** directly, because those are where most sessions start and end. Everything else is grouped, and the groups are named after questions rather than after subsystems — you do not go to "the audit subsystem", you go and find out who changed something.

3.1 Reading the header

Four things in the header are easy to miss.

Every view has an address. The address bar follows the view — `#/channels`, `#/messages`, `#/settings` — so a reload returns to it, a bookmark opens it, a link sent to a colleague lands them on it, and the back button moves between views rather than leaving the console. An address naming a view above your role opens the dashboard instead.

Search, or `⌘K`. It searches across channels, settings, messages and the manual, which means it is usually faster than navigating — particularly for a setting whose name you half remember.

Theme. Three of them: Midnight, Dark, and Light. Midnight is near-black and is what a wall display in a server room usually wants; Dark is a softer neutral grey; Light is for a laptop next to a window. The choice is stored in the browser rather than against the account, because a theme is a property of where somebody is sitting, not of who they are — the same operator wants Midnight on the wall and Light on a train. A machine's own preference is consulted once, to pick a sensible default the first time.

Manual. The document you are reading, served by the running server. It is the same file the binary can write to disk, so it always describes the version in front of you rather than the latest release.

3.2 Monitor — whether it is working, and what happened

3.2.1 Dashboard

The state of every channel, throughput, and anything currently wrong. This is the screen to leave open.

3.2.2 Messages

Every message that arrived, what it was transformed into, per destination, and what happened to it. Searchable, including by a filter expression, so "show me the ones from this sender that were rejected" is a query rather than an afternoon.

A message can be replayed from here, and reprocessed through the **current** configuration rather than the one that was live at the time — which is how you confirm a fix works against the traffic that broke it. See the message store for what is kept and for how long.

3.2.3 Queue

What is waiting to be delivered, and why. A queue that is not draining is the single most useful early warning an integration engine produces, and it is worth pairing with an alert rule so nobody has to be watching. See delivery.

3.2.4 Alerts

What has fired and what the rules are. Rules are edited here rather than in a file. See alerting.

3.2.5 Metrics

Throughput, latency and error rates over time. Useful for capacity questions and, more often, for board packs — which is why the site's own logo appears on it if one has been uploaded.

3.2.6 Flow map

A picture of how messages actually move: which sources feed which channels, and which destinations they reach.

Its value is not decoration. Every estate accumulates a channel nobody remembers commissioning and a destination nobody realised was still receiving, and both are visible here in a way they are not in a list of channel files.

3.2.7 Sharing what a feed actually contains

Profile this feed reports what is really arriving: which trigger events, which fields are populated and how often, what shapes the values take, and which codes a sender actually uses. That last one is usually the most useful single fact in it.

A profile can be downloaded and sent to somebody else. It needs a name and — required — which system produced the feed, because the question a stranger asks of a shared profile is whether it describes their system, and an optional field would make "Unknown" the commonest answer in any shared collection.

It carries no patient data, and that is checked rather than assumed. A profile is statistical: counts, rates, lengths, shapes. Values are kept only for fields the standard defines as code tables, because a code table value is not identifying. Identifier values are never listed.

Two things follow from that being a promise rather than a hope. The server rebuilds the profile from the stored messages when you export it, rather than sending the one on your screen — so what leaves the building was assembled by the profiler, which has a test proving this property, not by a browser. And the export refuses outright if a profile ever does list values for a field that is not a code table, naming the field but not the values.

A profile that arrives from elsewhere is checked the same way on the way in, for a different reason: it was produced by somebody whose care is unknown, and showing their patients' identifiers would be this server's disclosure. Importing one changes nothing — a profile describes a feed, and what to do about it is a separate decision.

3.3 Build — change what a channel does, and try it before it is live

3.3.1 Channels

The channel builder. Sources, destinations, filters, transformations, scripts and tests, with the generated YAML shown beside the form as you work — so the file is never a mystery, and anybody who prefers the file can read along.

See channels for what the parts are, and transformation for what the steps do.

3.3.2 Scripts

A workbench for scripts, separate from the builder because writing a script and wiring a script are different activities.

All seven slots are offered — filter, transformer, preprocessor, postprocessor, writer, lifecycle and reader — and choosing the right one matters, because the slot decides what the script receives and what it is expected to return. Checking a script as the wrong kind is not a workaround; it compiles a different thing.

JavaScript, Lua and WebAssembly are all available. See transformation.

3.3.3 Contracts

Expectations about what a feed must look like, created and checked here as well as from perfuse contract. See the command line.

3.3.4 Tables

Lookup tables and code sets: the mappings that turn one system's codes into another's. Editable here so that adding a code is not a deployment.

3.3.5 Mapper

Proposes a channel from a single sample message, and states explicitly what the sample cannot tell it. That second half is the part that makes it trustworthy — a proposal that looked complete would invite you to believe things the message never said.

3.3.6 Playground

A scratchpad for messages and transformations. Paste a message, apply steps, see the result.

Nothing is sent anywhere once the page has loaded, which is the reason it exists in this form: it is safe to paste something real into, on the understanding that it stays in the browser.

3.3.7 Shadow

Runs traffic through a candidate configuration alongside the live one and reports the differences, so a change can be evidenced before it is trusted. The same idea as `perfuse compare`, applied to a change rather than to a migration. See testing.

3.4 Exchange — other formats and other organisations

3.4.1 Team dashboards

A dashboard for each kind of person who watches the interfaces: Operations, Connections, Interface analyst, Prior authorization, Imaging, Revenue cycle, Laboratory, Privacy and security, and a plain-words Department summary. Tiles show counts, states and times, never a message or a patient, so a dashboard can go to a manager or a wall screen without widening who sees patient data.

Each person opens on the dashboard their directory group or role is given. `-dashboards` `dashboards.yaml` maps them:

```
groups: {PACS-Admins: imaging, Network-Team: connections}
roles: {viewer: manager}
```

A group wins over a role, and anyone unmatched opens on Operations. **Arrange tiles** hides and reorders tiles, the channel picker filters them, and **Save as my view** keeps that as your own dashboard until **Reset**.

Export to Grafana downloads the dashboard as Grafana JSON over `/metrics`, with a data source variable and a channel filter, for anyone who needs more than light configuration. Tiles with no metric behind them are left out, and the export's description names them.

The **Connections** tile checks every networked destination in layers. It looks the name up, opens a TCP connection, completes TLS where it is used, and reads the destination's last delivery and failure from real traffic. Each failure says what it means, for example "refused: the partner's engine is probably stopped", "timed out: a firewall or a VPN tunnel that is down", or "a certificate from an authority this server does not trust". Nothing is sent to the partner. With `-vpn tunnels.yaml` the **VPN tunnels** tile beside it reads each partner tunnel's state from AWS Site-to-Site VPN, an Azure VPN Gateway connection or strongSwan on this host, so a time-out can be told apart from a tunnel that is down. It also compares both sides' IKE and IPsec settings, as the cloud reports ours and the file lists theirs, and says where they cannot agree. **Connection sheet** downloads a partner's settings side by side as Markdown to send them; it never holds a pre-shared key. Figures a dashboard needs that are not recorded yet are listed under **Not measured yet** instead of being drawn as empty tiles.

The department dashboards compute their own figures on the server, and each one says what window and source it covers:

- **Prior authorization:** PAS requests decided within CMS-0057's 72 hours (expedited) or 7 days (standard), decided late, waiting,

and overdue now, with the median time to decide. Also how the services asked for were answered (approved, modified, denied, pending), as each stands now, over 30 days.

- **Imaging:** objects received by C-STORE per modality, and those not delivered on. Also the routing delay per channel (the median

and the 95th percentile), C-FIND queries and failures per query channel, and imaging reports (an ORU whose OBR-24 is an imaging section, or an MDM) that did not reach every destination.

- **Revenue cycle:** 999 transaction sets accepted, accepted with errors and rejected, and 277CA claims accepted and rejected

(A3, A4, A6, A7, A8). Also 835 claims matched by patient account number to 837s sent through this server, and the commonest rejection reasons.

- **Laboratory:** result delivery time, from receipt to the last destination's acknowledgement, and critical results (OBX-8 HH, LL,

AA, > or <) that did not reach every destination.

- **Privacy and security:** who opened messages, searched for patients or read prior authorizations in the console over 7 days, from

the audit log (opening a message is audited as `message.read`). Also each live API token and when it was last used; a token unused for 90 days is flagged.

3.4.2 FHIR lab

Convert HL7 v2 to FHIR, validate a resource, and inspect the result. The samples cover admissions, lab results, appointments, clinical notes, immunizations and a reportable visit. The validation includes US Core, which is the profile American regulation is written against, so this answers "is what they sent us actually conformant" without adopting anything.

Public health case report builds the eCR eICR a case reporting destination would send for the message. It shows which trigger code made the visit reportable, what the report had to mark as missing, and the document itself. A message with nothing reportable gets the reason instead. Nothing is sent from here.

3.4.3 Subscriptions

FHIR subscriptions and whether their notifications are arriving: events, accepted, waiting, and the last reason one was refused. Subscriptions are created by FHIR clients through the FHIR API, not from this page, and the credentials they gave the server are never shown. See message formats.

3.4.4 Documents

Clinical documents — CDA and the printable form of a document. Useful when the thing being exchanged is a discharge summary rather than a message.

3.4.5 Claims & auth

Build an X12 275 claims attachment in the 006020 form CMS-0053-F adopts, from a pasted C-CDA or an uploaded PDF, TIFF or JPEG, and see it read back. Read a 275 and download the document inside. Convert an X12 278 decision into a Da Vinci PAS ClaimResponse. Nothing on this page is stored or sent. See message formats for what the 275 is built from and what it does not do.

3.4.6 TEFCA

The audit trail for exchange through a Qualified Health Information Network, and the purpose-of-use rules that govern it.

Two transports, at genuinely different stages, and the screen says which is which. One flag covering both would be wrong in whichever direction it was rounded.

Facilitated FHIR is implemented, and has never spoken to a real QHIN. It follows the Sequoia Project Standard Operating Procedure effective 8 March 2026: discover the partner's UDAP metadata, verify it, register dynamically, obtain an access token carrying the purpose of use, and query. UDAP is a public key infrastructure over OAuth 2.0 — a trust community issues X.509 certificates to its members, and a member authenticates by signing JWTs with the private key and attaching the certificate chain.

The security layer is verified against somebody else's implementation, which is the only verification worth anything here. A registration signed by this build was sent to a live third-party reference server and refused with `unapproved_software_statement – Untrusted: Certificate is not a member of community`. To produce that answer the server decoded the request, parsed the software statement as a JWT, checked its signature, walked the certificate chain, and reached a decision about membership. It is the difference between being turned away at the door and being handed back an illegible letter.

What remains cannot be done from here. The certificate that proves community membership comes out of QHIN onboarding and is not something software produces. Until one exists, no exchange with a real partner completes.

The older QHIN-to-QHIN exchange built on the IHE profiles is not implemented. An attempt is refused with an error saying so, and the refusal is audited as a failure. That is deliberate rather than pending a tidy-up: a stubbed exchange reporting success would put entries in the audit trail for exchanges that never happened, and a trail that disagrees with what happened is worse than none, because the trail is what gets believed.

Throughout, the endpoints used for exchange come from the **signed** part of a partner's metadata, never the document body. Verifying a signature and then reading the endpoints from the unsigned copy would be an elaborate way of trusting the network: an attacker able to rewrite the response leaves the signed element alone, points the token endpoint at their own server, and collects client assertions from everybody who checked the signature and then ignored what it covered.

A trust anchor bundle is required and its absence is refused at startup. Checking a partner's signed metadata against the certificate that arrived inside it proves only that one party made both — a mistake this codebase has made before, in SAML, where the verifier preferred the embedded certificate and authenticated an attacker as an administrator while every test passed. Include the intermediates in the bundle: a UDAP server need only send its leaf certificate, and the public reference server does exactly that.

3.5 Administer — who can do what, and how this server is set up

3.5.1 Users

Accounts, roles, and what each role is allowed to do. See security.

Machine credentials are also here. A token is for a machine rather than a person: it does not expire and is not affected by anybody signing out, which is what a server polling its neighbour needs. Only a hash is stored, so a token is shown once when it is created and cannot be recovered — a stolen database yields no usable token. Give one the lowest role that does the job.

3.5.2 Certificates

The TLS certificates in use and when they expire. The expiry dates are the point: a certificate that lapsed on a Sunday is a class of outage that is entirely preventable and routinely is not prevented.

3.5.3 Activity

Who changed what, and when. Every configuration change is recorded, which is what makes "it started failing on Tuesday" answerable.

3.5.4 Fleet

Every Perfuse instance on one page.

Mirth sells this as a separate product; here it is a configuration file and a read-only token. The reason it is worth having is narrow and important: **a dead server does not look like a quiet one**. A single instance's dashboard cannot tell you that a second site stopped reporting, and a fleet view can.

3.5.5 Migrate

Drop in a Mirth or OIE channel export and see how many of your channels would run here unchanged. The same analysis as `perfuse explain` and `perfuse translate`, without needing a terminal. See migration.

3.5.6 Settings

Every setting the server has, with a control for each — including the ones that would traditionally live only in a file. Each says whether it takes effect at once or needs a restart, and whether it has been set explicitly or is sitting at its default.

Branding is here too: a site can set the product name, a tagline, an accent colour, and upload a logo. The logo then appears in the header, on the sign-in page, on the dashboard and on the reporting screens. If none is uploaded, nothing is substituted — an unbranded installation stays unbranded rather than acquiring decoration.

3.6 What the colours mean

Four registers, and each says something different. The distinction is worth stating because it is the difference between a screen you can read at a glance and one where everything competes for attention.

Plain text is a fact about how Perfuse works. Most of the interface. Nothing is coloured to tell you it is important.

Green means something succeeded just now, because of something you did. It appears after an action and does not persist.

Amber is a caution: worth knowing before you act, and not a fault. A feature that is not switched on, a setting whose changes are audited, a conversion that approximates rather than reproduces, a transport Perfuse handles but cannot fully verify. Amber is the colour of "read this before you continue".

Red means something is wrong at this moment. A value that was rejected, a request that failed, a certificate that has expired, a critical alert that is firing. Nothing else uses it.

That last rule is why a screen does not open red. Red is the only colour that carries an instruction — stop, and it may be something you did — and a screen that spends it on decoration has nothing left to say when the input really is wrong. This was not always true: the users screen used to open with "the server sent a response that could not be read" on every installation that had not configured passkeys, and the settings screen marked audited values in the same colour as a rejected one. A test in `web/src/RedMeansWrong.test.ts` now fails if a phrase describing something absent or approximate is given a red style.

Inside a code box the vocabulary is different. There, red is a token type — an HL7 segment name, an XML element — and carries no judgement at all.

4 Channels

A channel is one flow of messages: somewhere they come from, optional changes, and one or more places they go. It is the unit of configuration, the unit of monitoring and the unit of deployment. Everything else in Perfuse exists to serve channels.

4.1 Anatomy

A channel file has these parts, in the order they take effect:

Part	Key	Required	What it does
Identity	name	yes	Names the channel. Must be unique.
Data type	dataType	no	How to parse messages. Defaults to HL7 v2.
Source	source	yes	Where messages arrive from. Exactly one.
Filter	filter	no	Decides which messages to accept.
Transformations	transformations	no	Ordered changes to the message.
Destinations	destinations	yes	Where messages go. One or more.

Plus several that apply across the whole channel: scripts, shadow, contract, tables, attachments, group and enabled. Each is in the channel reference.

4.2 One source, many destinations

This asymmetry is deliberate and it is worth understanding early, because it is the single most common source of confusion when coming from a different engine.

A channel has exactly one source. If messages arrive from two places, that is two channels. Trying to express it as one channel produces a configuration where a failure in either transport looks the same from outside, and where the two feeds' traffic is mixed together in the statistics — so a drop in one is hidden by the other.

A channel has any number of destinations, and they are independent. One failing does not stop the others, and each has its own queue, its own retry policy and its own delivery record. A message that reached three of four destinations is recorded as exactly that, not as a success or a failure.

4.3 Destinations are not a chain

Each destination receives the message as it stood after the channel's transformations. A destination does not receive the output of the destination before it.

This matters when a destination has its own transformations, which it may. Those apply to that destination's copy only. A common mistake is to put a change on the first destination and expect the second to see it.

If you genuinely need a chain — the output of one flow feeding into another — use a channel destination, which sends to another channel by name. That is explicit, appears in the message history as two channels, and can be monitored as two things, which a hidden chain cannot.

4.4 Naming

The name is the channel's identity everywhere: in the file, in the message store, in alerts, in the audit log and in the URL. Changing it is not a rename; it is a new channel, and the old one's history stays under the old name.

group is separate and exists for organising the interface. A group has no effect on behaviour, and grouping channels does not make them share anything.

On a multi-facility deployment, put the facility in the name rather than only in the group. Mirth deployments across several sites reliably drift into inconsistent naming, and the point at which that becomes painful is an incident at two in the morning where the channel list has four things called `adt-inbound`.

4.5 Enabling and disabling

`enabled: false` stops a channel starting without deleting it. Its configuration, history and statistics stay.

A disabled channel is not a paused one. Its source is not listening, so a sender attempting to connect gets a refused connection rather than a hung one. For MLLP that is the right behaviour — the sender will queue and retry — but it does mean the sender's own error log will fill up, so a disabled channel is not a substitute for arranging an outage with whoever is sending.

4.6 Where channels live

One YAML file per channel, in the directory given by `-channels`. The filename is not the channel name; the `name` key is. Keeping them the same is strongly advised and nothing enforces it.

Files are re-read without a restart. A file that does not parse is reported and the previously loaded version keeps running, so a syntax error cannot take a working feed off the air.

A channel edited in the web interface is written back to its file. If those files are in version control — which is recommended — expect commits from configuration changes made through the interface, and do not hand-edit a file at the same time as somebody has it open in the builder.

4.7 Encrypting a channel's own listener

A channel that listens on a port is its own server, and its TLS is separate from the TLS on Perfuse's web interface. Setting one does not set the other. For an HTTP source the settings are under **TLS and limits** on the source.

Switching TLS on asks for a certificate and a private key. Requiring a client certificate turns it into mutual TLS: the sender must present one signed by an authority you name, which is authentication, so a listener with it on does not also need a shared token — though having both is not wrong.

The authority is asked for only once a client certificate is demanded, and is left out of the file until then. A channel naming an authority that is never consulted reads as though senders are being verified when they are not.

An MLLP listener has the same settings under **Encrypt this listener**, with one difference worth stating: on MLLP a client certificate is the only authentication available. There is no token and no header, so a listener without one accepts messages from any host that can reach the port.

Two limits sit alongside it. **Give up reading after** a duration, because a sender that opens a connection and stops writing otherwise holds it open. And **largest message accepted**, in bytes, because zero means no limit and the limit is then memory — a sender that posts a gigabyte is refused rather than absorbed.

4.8 Validation

A channel is checked when it loads, and the checks are refusals rather than warnings where the alternative would be a channel that runs and does the wrong thing.

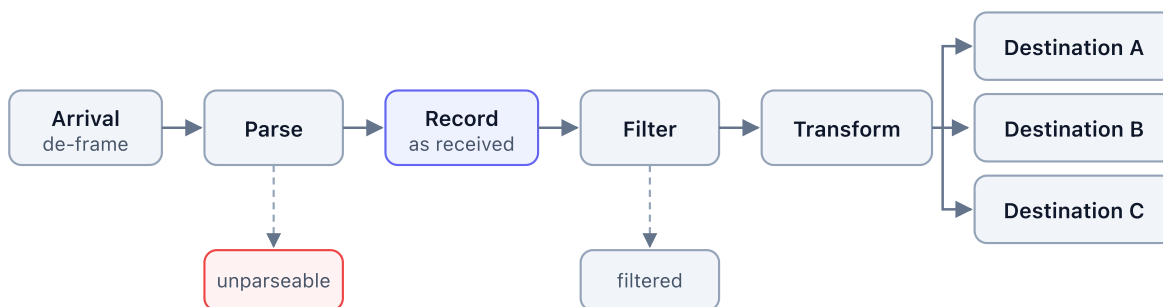
Examples of things refused outright: a destination with no type, a transformation addressing a path that cannot exist in the declared data type, an acknowledgement mode that the source transport cannot deliver, and a Schedule II prescription with refills.

The distinction Perfuse tries to hold to is that anything which would produce a plausible-looking wrong result is refused, and anything which is merely unusual is allowed and reported. A channel that cannot be wrong is more valuable than a channel that starts.

5 Message Flow

This chapter follows one message from the moment it arrives to the moment it is delivered or given up on. Knowing this order is the fastest way to work out where a change belongs, and why something is not happening.

5.1 The order of events



Filtered is a normal outcome, not an error. Unparseable is recorded and stops.

The acknowledgement is sent after delivery by default, so a sender is not told "received" before it is true.

1. **Arrival.** The source accepts the bytes. For a stream transport this includes de-framing — finding where one message ends and the next begins.
2. **Parse.** The bytes become a tree according to the channel's dataType. A message that cannot be parsed stops here and is recorded as a parse failure.
3. **Record.** The message is written to the message store as received, before anything modifies it. This is what makes replay and reprocessing possible.
4. **Filter.** If the channel has a filter, it decides whether to accept the message. A rejected message is recorded as filtered, which is a normal outcome and not an error.
5. **Transform.** The channel's transformations run in order, each against the result of the last.
6. **Fan out.** Each destination gets a copy of the transformed message, applies its own transformations if it has any, and delivers.
7. **Record the outcome.** Per destination: delivered, queued, failed, or filtered.
8. **Acknowledge.** If the source expects an acknowledgement, it is generated and sent. By default this happens here, after delivery — see below.

5.2 When the acknowledgement is sent

This is the most consequential setting in a channel, because it decides what the sender is being promised. It is `ack.when`, and it has two values.

`on_delivery` is the default. The acknowledgement is sent only after every enabled destination has accepted the message. A positive acknowledgement therefore means the data actually arrived somewhere, which is the strongest claim an engine in the middle can honestly make.

`on_receipt` acknowledges as soon as the message is parsed and queued, before any destination has been written.

Perfuse defaults to `on_delivery` because promising delivery and then losing the message is worse than being slow. Under `on_receipt`, a message that has been acknowledged can still be lost if the process dies with work queued — the sender believes it is delivered, nothing upstream will resend it, and the only record that it existed is in Perfuse's own store.

5.3 When to choose `on_receipt`

The case for it is real, and it is timeouts. An MLLP sender that does not get an acknowledgement within a few seconds will typically resend, so a slow destination under `on_delivery` produces duplicates rather than delay. If one destination is a slow archive or a system that is regularly unavailable, `on_delivery` couples the sender's timeout to that destination's worst case.

The trade is explicit: `on_receipt` moves the risk from duplicates to loss. Choosing it means accepting that an acknowledged message can be lost in a crash, and it is worth pairing with queue-depth alerting so that a queue which has stopped draining is noticed. See alerting.

Under either setting, a positive acknowledgement is not a promise that the receiving application has *processed* the message — only that it accepted the bytes. If you need to know a result was filed, that has to come from the receiving system, and no integration engine can provide it.

5.4 Where the message is recorded

Before the filter, and therefore before any transformation. Three consequences:

- A message the filter rejected is still in the store. You can see what arrived and confirm the filter was right.
- Replay and reprocessing always start from what actually arrived, not from a partially transformed version.
- A change to the transformations can be tested against real historical traffic, because the original is still there. See testing.

The transformed form is also recorded, per destination, alongside the outcome. So the store holds what arrived, what was sent, and what happened — which is what an investigation needs.

5.5 Failure at each stage

Stage	What failure looks like	What happens
Arrival	Framing error, connection dropped mid-message	Recorded, connection closed, sender retries
Parse	Not valid for the declared type	Recorded as a parse failure, negative acknowledgement if the source expects one
Filter	The filter itself errors	Treated as a refusal, recorded with the error
Transform	A step fails	Depends on the step's <code>on_error</code> ; see transformation
Deliver	Destination unreachable or rejects	Queued and retried per the destination's policy; see delivery

The important asymmetry: a parse failure is permanent and a delivery failure usually is not. Retrying an unparseable message will never help, so it is not retried. Retrying an unreachable destination usually will, so it is.

5.6 Ordering

Within one destination, messages are delivered in the order they arrived, and a message that fails and enters the queue holds its place — later messages wait behind it rather than overtaking.

This is the right default for clinical data, where an admission followed by a discharge delivered in the wrong order is worse than both being late. It also means one poisoned message can hold up a queue, which is why queue depth is worth alerting on.

Across destinations there is no ordering guarantee, because they are independent. Two destinations will not necessarily receive the same message at the same time, and a fast one may be several messages ahead of a slow one.

Ordering across channels is not guaranteed at all. If two feeds must be applied in a particular order relative to each other, that ordering has to be enforced by the receiving system, because nothing in the middle can know which of two independently arriving messages happened first.

5.7 Concurrency

A source may handle several connections at once, bounded by `max_connections`. Messages from different connections are processed concurrently.

This means the arrival order of two messages sent simultaneously down two connections is not defined, and cannot be — they genuinely arrived at the same time. A sender that needs ordering has to use one connection, which is what almost all of them do.

6 Transformation

Transformations change a message. They are a list, they run in order, and each one sees the result of the one before it.

Perfuse's transformations are declarative: each step names an operation and the field it applies to. There is no scripting language in the ordinary path. This is a deliberate limitation and the reasoning is in why not a script below.

6.1 The steps

Step	What it does
set	Writes a literal value to a path, creating it if absent.
copy	Copies one path to another, with an optional default when the source is absent.
clear	Empties a field, leaving it present.
remove	Removes a field entirely.
map	Looks the current value up in a table and replaces it.
replace	Regular expression substitution within a field.
pad	Pads a value to a fixed width.
date	Converts a date or timestamp between formats.
trim	Strips surrounding whitespace.
case	Converts to upper or lower case.

Every step also accepts `description`, which appears in the trace and in the interface, and `when`, which makes the step conditional.

A step is written as the operation name with its own block:

```
transformations:
- description: Our record number, not theirs
  set:
    path: PID-3.4
    value: RIVERSIDE
- copy:
    from: PID-19
    to: PID-3.1
    default: UNKNOWN
- map:
    path: PID-8
    table:
      M: Male
      F: Female
```

set and value are not siblings. set takes a block containing path and value. Writing set: PID-3.4 with value: beside it is the commonest mistake in a hand-written channel and produces a load error rather than a silent misconfiguration.

6.2 Clear, remove and the difference

c`clear` leaves the field present and empty. `remove` takes it out altogether.

For HL7 this distinction is real and it reaches the receiver. An empty field says "I have this field and it has no value". An absent field says nothing at all. Some receiving systems treat the first as an instruction to blank their stored value and the second as an instruction to leave it alone — which is the difference between deleting a patient's recorded allergy and not mentioning it.

If you do not know which the receiver wants, ask. Guessing has a fifty per cent chance of silently destroying data on the far side.

6.3 Conditions

`when` takes an expression and the step runs only if it is true:

```
- when: MSH-9.2 == "A08"
  set:
    path: EVN-1
    value: A08
```

A step whose condition is false is recorded as **skipped**, and this is reported separately from a step that ran and changed nothing. That distinction is the point of the step-through debugger and it is covered in debugging: a false condition and an absent field look identical from outside and mean opposite things.

6.4 Paths

A path addresses a field: PID-5 is the fifth field of PID, PID-5.1 its first component, PID-3 [2] the second repetition. Without a repetition index a path addresses the first repetition.

Two behaviours worth knowing:

- `set` creates whatever path it addresses, including intervening structure. This is convenient, and it means `set` can never demonstrate that a field was absent — after the step it exists. To test what happens when a field is missing, use `copy` from a path that is not there.
- A path addressing a segment that occurs several times, without an index, addresses the first occurrence. On a message with five OBX segments, OBX-5 is the first one's fifth field. To reach all of them the path needs an index, and a channel that assumes one OBX will silently ignore the rest.

6.5 Errors

Most steps have nothing to fail at. Setting a value, clearing a field or copying from an absent path with a default are all defined for every input.

`date` is the exception, because a value that is not a date it recognises has no correct conversion. It takes `on_error`:

- `fail` — the default. The message stops and is recorded as a transformation failure.
- `keep` — leave the original value alone.
- `clear` — empty the field.

The default is `fail` because a timestamp that silently did not convert is worse than a message that did not arrive. A date in the wrong format usually still looks like a date, so it passes the receiver's validation and lands in the record as the wrong day.

`keep` is legitimate when the receiver tolerates either format, and when used the trace marks the change as deliberate so it does not read as a step that failed quietly.

6.6 Order matters, and the trace shows it

Each step sees the previous step's output. Two consequences that catch people:

- A `map` after a `case` sees the case-converted value, so its table keys must match the converted form. A table of M/F after a lower-case step matches nothing, silently, because a value not in the table is left alone.
- A `copy` from a field that a later step removes still works, because it ran first. The reverse does not.

Both are visible in the step-through view, which shows the whole message after each step rather than only the change. See debugging.

6.7 Suggesting mappings, and what to do with a suggestion

The **AI Mapper** proposes mappings between a set of source fields and a set of targets. Its defining behaviour is that it declines: below its confidence threshold it abstains rather than guessing, because a confident wrong mapping in a clinical system is worse than no mapping at all.

6.7.1 Give it something to be confident about

A name on its own is weak evidence, even when it matches exactly. PatientMRN against PatientMRN scores 68, which abstains at the default threshold of 70.

The engine knows what an HL7 path means — it will tell you PID-7 is Date/Time of Birth and score your field name against that rather than against the string PID-7, which shares no characters with anything. A locally defined Z segment means whatever your site decided, so nothing is claimed about it.

That gets a mapping most of the way. To finish it, supply both halves:

- **Example values** for a source field, after an equals sign: ZPI-1 = MRN00412, MRN00998
- **The kind of value** a target holds, the same way: PID-3.1 = mrn. The kinds are mrn, npi, ssn, phone, date and email.

With those, the engine can see that the values agree with the target's kind, and a suggestion rests on more than a name.

6.7.2 Approval is one mapping at a time

Each suggestion has its own approval box. There is no "accept everything above 80%", deliberately: the engine has already used the confidence to decide whether to offer the mapping, and what remains is a judgement about this field in this feed. A number is not that judgement.

An abstention cannot be approved at all. It has no box, not a disabled one, and the screen says the engine does not know. If you know the answer, map it by hand — that is a different act from agreeing with a suggestion.

A suggestion below your threshold is shown without a box too, marked as below it. Worth reading — a near miss is often the mapping you wanted, and hiding it would leave you wondering whether the engine considered it — but not approvable, because your threshold is the line you drew.

6.7.3 Two things to do with what you approved

Add them to a channel turns each mapping into a copy step. This only works where the source is a path in the message, such as ZPI-1, because a step copies from one place in a message to another. The steps go into the channel file and take effect when it next loads — nothing changes in the traffic until then. Each step's description records the confidence it was suggested at and that a person approved it, which is what makes it accountable six months later.

Download as a recipe is the right answer for the commoner case: a mapping from a column name, a JSON key or a vendor's own field name. Those are real mappings, and a channel cannot read such a field directly, so a step cannot express them. The recipe records them as approved and can be shared.

6.8 Why not a script

Mirth puts a JavaScript transformer at the centre of every channel, and it is the most flexible design available. Perfuse offers scripting — see the `scripts` key in the channel reference — but does not make it the ordinary path, for three reasons that come from what goes wrong in practice.

A declarative step can be inspected. Perfuse can tell you that a step reads PID-5 and writes PID-8, before running it, and can therefore show a field-level trace, warn that a step addresses a path the declared data type cannot contain, and generate a per-step replay. None of that is possible for arbitrary code.

A declarative step can be reasoned about by the interface. The builder can offer the fields that exist in this channel's traffic, and a change to a step can be checked against real historical messages before it is saved.

And a declarative step cannot do something unrelated to the message. A script can call out to the network, read the filesystem, or depend on the time of day. When a channel that worked for two years starts producing wrong output, the cause being outside the message is the hardest case to diagnose, and removing the possibility is worth more than the flexibility it costs.

Where a genuine transformation cannot be expressed declaratively, use a script and accept that the trace will show it as one opaque step.

6.9 What a script is allowed to do

A script is refused anything outside the message unless it is granted, and the grants are on the Scripts section of the builder. This is deliberately unlike Mirth, which gives every script all of them — so there, a transformer can quietly start reading the filesystem after a copy-paste.

There are three.

Read and write files requires you to name the directories it may use, and a channel granting file access without them is refused at load. That requirement is the whole difference between file access and unrestricted file access: granting it without directories once reached the entire filesystem, including Perfuse's own database, which holds password hashes, the LDAP service account password and the OIDC client secret. Creating a channel needs only the editor role, so the escalation completed the moment an administrator started the channel.

Query a database uses the connections configured on the channel. A script with this can read anything those credentials can.

Route messages to other channels lets a script send a message into another channel, which is how a loop between two channels becomes possible.

Granting a capability to a channel with no scripts does nothing, and the generated file omits the whole block in that case.

6.10 Handling a date that does not match its format

A date step converts from one format to another, and the interesting question is what happens when a value does not match the format it claims.

The default is to **stop the message**, and it is the right default: a timestamp that silently did not convert is accepted downstream and then misread, which is worse than a message that failed loudly and is sitting where somebody can see it.

Leave the value as it is passes it through in its original format. Downstream then sees two formats in one field, which is the failure the step existed to prevent — but it is the right choice where the field is optional and a partner sends it inconsistently.

Empty the field is honest and better than a wrong format, but a receiver that requires the field then rejects the message instead.

6.11 The scripting languages

Three are accepted, and which one to pick depends mostly on whether you are migrating. The value goes in a script's `language` key:

Value	What it is
<code>javascript</code>	The default. An unmarked script is JavaScript.
<code>lua</code>	The better choice when nobody is migrating.
<code>wasm</code>	The bytes of a compiled WebAssembly module, not source.

JavaScript is the default, and an unmarked script is treated as JavaScript. It is here as a compatibility obligation rather than a preference: Mirth's scripts are JavaScript and a site moving across has thousands of lines of it. That includes support for E4X, a long-dead extension to the language that Mirth scripts use for XML and that has to be translated before anything will parse it.

Lua (`lua`) is the better choice for anyone not migrating. A smaller runtime, no prototype chain to misuse, and integer-indexed tables that match segment and field addressing more naturally than JavaScript's do. Its standard library is small enough to enumerate, which is what makes its sandbox easier to be confident about.

WebAssembly (`wasm`) is not a language but a compiled module, and the field wants the bytes of a `.wasm` file rather than source. Use it when the transformation already exists in a compiled language, or when it is heavy enough that the runtime cost matters.

Its contract is deliberately the simplest one available: **a module reads the message on standard input and writes its result to standard output**. Bytes in, bytes out — which is what every toolchain already does, and what somebody testing a module on their own command line already has. The alternative was an exported function over linear memory, which is faster and requires every module author to agree

about allocation, string encoding and who frees what; for something that runs once per message, that trade is not worth it.

Anything a module writes to standard error becomes log lines, so it can explain itself the way a script calling `logger` can.

A non-zero exit means the module *failed*, which is not the same as a filter deciding to reject a message. A filter module says `true` or `false` on standard output. Conflating the two would make a crashing module indistinguishable from one deliberately dropping traffic.

6.11.1 What the output means

The rules are the same for all three languages rather than new ones per runtime, so a script's observable behaviour — what it can see, what it may do, how long it may take, what a filter verdict means — is defined once.

For a transformer, preprocessor or writer, **empty output means the message is unchanged**. Falling off the end of a script leaves the message alone; treating no output as "the message is now empty" would discard traffic on a technicality.

WebAssembly modules are compiled when the channel loads, not on the first message. Compiling is not free — a module built from a language that brings its own runtime can take hundreds of milliseconds — and paying that at load time means the first message through a channel is not mysteriously slower than the rest.

7 Filtering

A filter decides which messages a channel accepts. It is one expression on the channel, evaluated after the message has been parsed and recorded, before any transformation.

```
filter: MSH-9.1 == "ADT" && MSH-9.2 in ("A01", "A03", "A08")
```

A message the filter rejects is recorded with the outcome **filtered**. That is a normal result, not an error, and it does not raise an alert or produce a negative acknowledgement.

7.1 Filtered is not failed

This distinction is worth being firm about because conflating them is how a real problem gets hidden.

A filtered message was correctly declined: it was not for this channel. A failed message was one this channel should have handled and could not. If both were recorded the same way, a channel that starts failing every message would look exactly like a channel that is filtering most of its traffic, which is the normal state of many channels.

So the statistics count them separately, the alert rules treat them differently, and the interface shows them in different colours. A rise in filtered messages might mean the sender has started sending something new; a rise in failures means something is broken.

7.2 What a filter can read

Any field in the parsed message, by the same paths transformations use. The filter runs before transformations, so it sees the message as it arrived — not as it will be sent.

That ordering is deliberate. A filter that read post-transformation values would be deciding whether to accept a message based on changes made on the assumption it was accepted, which is circular and produces channels whose behaviour depends on the order of two things that look independent.

7.3 Expressions

Comparisons: ==, !=, and for numbers <, <=, >, >=.

Membership: in ("A01", "A03"), which is clearer than a chain of || and is the form the builder produces.

Combination: &&, ||, and parentheses. Negation with !.

Absence: a path that is not present compares unequal to any value, including the empty string. To test for absence explicitly, compare against absent. This matters because an absent field and an empty field are different, as described in transformation, and a filter that treats them the same will accept messages it was written to exclude.

7.4 An error in the filter is a refusal

If the filter itself fails to evaluate — a malformed expression that got past validation, or a comparison that cannot be made — the message is refused and the error is recorded against it.

Refusing rather than accepting is the safer of the two, but neither is good, and the outcome is recorded distinctly from an ordinary filter refusal so it can be alerted on. A filter that is erroring on every message is silently dropping the entire feed, and it must not look like a filter that is working.

7.5 Filter at the source where you can

A filter runs after the message has crossed the network and been parsed. Some sources can filter earlier and more cheaply:

- A broker source takes a selector, which filters at the broker. Messages this channel does not want never cross the network at all. On a shared queue that is the difference between reading a hundred messages a day and a hundred thousand.
- A database source's query decides what it returns.
- A file source's pattern decides which files it picks up.

Filtering at the source is a performance decision, not a correctness one, and it has one cost: a message filtered at the broker is never recorded by Perfuse, so it does not appear in the message store and cannot be inspected later. If you need to be able to prove what the sender sent, filter here rather than there.

7.6 Testing a filter

A filter can be checked against real historical traffic before it is saved. The interface reports how many of the last N recorded messages the filter would accept, reject, and error on — which is the fastest way to find out that an expression which looks right accepts nothing.

The trace also shows, per message, which fields the filter read and what values it found. A filter that unexpectedly rejects everything is usually reading a path that does not exist in this feed, and the trace names it.

8 Message Formats

A channel's `dataType` decides how bytes become a message. It affects parsing, what paths mean, how the message is written back out, and which transformations are valid.

Value	Format
hl7	HL7 v2, the pipe-delimited standard. The default.
hl7v3	HL7 v3 and CDA documents, which are XML.
x12	ASC X12, used for eligibility and claims.
ncpdp	NCPDP Telecommunication Standard D.0 pharmacy claims.
script	NCPDP SCRIPT, electronic prescriptions. XML.
dicom	DICOM imaging objects.
delimited	CSV and other delimited text.
raw	No parsing. Bytes in, bytes out.

8.1 HL7 v2

The default, and what most of this manual's examples use. Segments terminated by carriage return, fields by `|`, components by `^`, repetitions by `~`.

Two things about the encoding are worth knowing because they cause silent faults.

The delimiters are not fixed. They are declared in MSH-1 and MSH-2 of each message, and Perfuse reads them from there rather than assuming pipes. A sender that uses a different field separator parses correctly.

The consequence is that a message whose separators have been replaced — most often by a word processor or mail client substituting typographic look-alikes — parses perfectly into fields that are all wrong. Nothing errors. This is why the sample reader described in testing reports which separator character it found, and refuses to express confidence when it is unusual.

When somebody sends you a sample and the channel built from it produces nonsense, check the separators before anything else. Ask for a plain text attachment rather than a message pasted into an email body.

8.2 HL7 v3 and CDA

XML rather than delimited. Paths are element and attribute paths rather than field positions.

In v3 a value is almost always an attribute rather than element text. A comparison that only reads element text will report two documents as identical when the birth date, the gender and every identifier differ. Perfuse's comparison reads both, and this is why: the alternative silently succeeds.

hl7v3 options are in the format reference.

8.3 X12

Segments terminated by ~ conventionally, with delimiters declared in the ISA segment. Like HL7, they are read from the message rather than assumed.

X12's structure is envelopes within envelopes — interchange, functional group, transaction set — and a single interchange can hold many transactions.

By default an interchange is **one message**. The whole file gets one record, one outcome and one row in the interface, however many transactions it contains.

8.3.1 Splitting an interchange

Setting `split` makes each transaction set a message in its own right, with its own record, its own metrics and its own row. An operator looking for one claim then finds one claim rather than a file of four hundred.

It is off by default because turning it on multiplies everything downstream — message counts, rows, alert thresholds — and a threshold tuned for files that is suddenly counting claims will not mean what it used to. That should be a decision somebody made rather than a default they inherited.

`split` and `acknowledge` cannot both be set, and a channel with both is refused at load. An acknowledgement is a statement about a whole interchange, so splitting would send one per transaction set, and a partner receiving several 999s naming the same functional group has no way to reconcile them.

8.3.2 When the counts do not add up

X12 declares its own counts — how many segments, how many transactions — and the `envelope` key says what to do when they disagree with what is actually present. Those counts are the only mechanism X12 has for detecting that half a claims file arrived.

Value	Behaviour
require	Refuse the interchange. The default.
warn	Process it, and record the fault against the message.
ignore	Do not check at all.

warn exists for a real situation: some partners' software has generated wrong counts for years in otherwise complete files, and a site that has to process those needs a way to say so out loud in the configuration rather than turning validation off wholesale.

ignore is named to be uncomfortable. Setting it throws away the only means X12 gives you of noticing a truncated file, and a truncated claims file is indistinguishable from a small one.

8.3.3 Transactions Perfuse reads as structured data

Any X12 interchange can be routed, filtered and addressed by path. These transactions are additionally parsed into named fields, so a channel can ask what a payer decided rather than which element sits at position four.

Transaction	What it carries
835	Remittance advice: what was paid, what was denied, and the adjustment reasons.
275	Claims attachments: clinical documentation supporting a claim.
277	Claim status, including a payer's request for additional documentation.
278	Prior authorisation: the request for review, and the payer's decision.
999	Functional acknowledgement, which Perfuse also generates.

271	Eligibility: whether a patient is covered, and their co-payment, coinsurance, deductible and out-of-pocket amounts, in and out of network.
834	Enrolment: members added, changed and terminated, with each coverage line and its dates.
837	Claims, professional and institutional (read for CMS-0057's CARIN conversion).

Perfuse also **builds** the 270 eligibility inquiry and the 276 claim status request, under **Exchange → Claims & auth** or through POST `/api/x12/eligibility/build` and `/api/x12/claimstatus/build`. Both are the HIPAA 5010 versions (005010X279A1, 005010X212), read back and envelope-checked before

they are shown, and neither is sent from that page: delivering one is a channel's job, where the trading partner and its credentials are.

8.3.4 Eligibility, and the CAQH CORE data content rule

A 271 is a list of EB segments, each a single fact: active coverage, or a \$25 co-payment for service type 30 in network, or \$800 of the individual deductible remaining. The reader assembles them into the answer a front desk wants - "coverage active under Gold PPO 2000; co-payment in network \$25; individual deductible remaining in network \$800" - and names a rejection (an AAA segment) in words, with what the payer says to do next.

It also checks the response against the CAQH CORE Eligibility & Benefits data content rule: for an inquiry about health benefit plan coverage (service type 30), an active response should carry plan dates, the plan name, co-payment, coinsurance, deductible and remaining deductible both in and out of network, and the base service types (1, 33, 35, 47, 48, 50, 86, 88, 98, AL, MH, UC). Each is listed as met or missing. This is Perfuse's reading of the rule, useful for "why does this payer never tell us the deductible?" - it is not CORE certification, which CAQH's authorised testing vendors carry out.

8.3.5 Prior authorisation

Perfuse implements prior authorisation twice, on purpose.

The **FHIR** side is what CMS-0057-F requires payers to expose by 1 January 2027. The **278** is what payers accept today, and the rule does not replace it — Da Vinci PAS is in practice a mapping onto this transaction, because a 278 is what a utilisation management system speaks. An engine reading only the FHIR half can talk to the systems that will exist and not to the ones that do.

A 278 response carries the **authorisation number**, and that is the field the whole transaction exists to deliver. Without it a certified service is not billable: the claim goes in with no authorisation number and is denied for the lack of one, which reads downstream as a clinical denial.

A denial and a refusal to consider are not the same thing

This is the most expensive confusion in the transaction, and Perfuse reports the two separately.

What arrived	What it means	What to do
HCR with action A3	The payer considered the request and will not certify it.	Appeal, or change the plan of care.
An AAA segment	The payer would not consider it — patient not found, requester not recognised, review not covered. Nothing was decided.	Correct the request and send it again.

Both read as "not approved" to anybody scanning, and the correct actions are opposite. Appealing something nobody ruled on achieves nothing; resubmitting a denial produces the same denial. The parsed outcome names which happened — denied against not-considered — and AAA04 carries the payer's own advice on whether the request is recoverable at all.

Partial certification is reported as partially-certified rather than folded into approval. There is an authorisation number to bill against, so it counts as approved, but the quantity certified is not the quantity requested — and a practice booking six sessions against three approved discovers it at the fourth claim.

An action code Perfuse does not recognise is reported as unknown, not as pending. Pending reads as "wait", and waiting is the wrong action for most of what an unrecognised code could be.

A channel can route on the decision directly:

```
filter: HCR-1 == "A3"
```

Converting between the two

Perfuse converts a parsed 278 into a Da Vinci PAS `ClaimResponse`, and reports every judgement the conversion needed rather than only the result. The conversion follows three rules:

- **No decision is upgraded.** Partial certification stays partial, and a request the payer refused to consider does not become a denial. Both collapses are available in the obvious mapping and both are wrong in the expensive direction.
- **Nothing is invented to fill a required field.** A certified event with no authorisation number is reported as such, not given a plausible one — a fabricated number produces a claim that is denied later for a reason nobody can trace.
- **A decision carried only at line level is used.** A payer answering per service can leave the event without a decision of its own; reading only the event level would report that as pending, and a practice waits for an answer it already has while the approval expires. Where the lines disagree, the least favourable wins, because an event with a refused line is not an approval.

Narrowing to a three-state vocabulary — approved, denied, pended — is available and explicit about what it discards. Narrowing a refusal to consider yields "denied" and says in words that the correct action is a corrected resubmission rather than an appeal.

Checked against PAS 2.2.1, not only against itself

The `ClaimResponse` is shaped to the Da Vinci PAS 2.2.1 profile and was checked with the official HL7 validator loaded with the published PAS package. The first version had only been checked by Perfuse's own tests, and the validator rejected it with six to eight errors per resource. Every one of them is something a payer's PAS endpoint would also have rejected:

What was wrong	Where PAS puts it
The decision was in adjudication.category and reason.	The reviewAction extension, on an adjudication whose category is submitted.
The reason code named X12 element 1034.	X12 external code list 886.
A pending request answered outcome: queued.	outcome is about processing and PAS allows only complete, error and partial. A pend is complete, with review action A4.
processNote.type was a CodeableConcept.	R4 has a code: print.
patient, created, insurer and request were missing.	All four are required. They are given as identifier references, because the response stands alone rather than in a bundle.

The corrected output validates with no errors for a certification, a denial and a pend. The warnings that remain are the validator saying it cannot load X12's code systems, which X12 licenses. The authorisation number is carried in reviewAction.number, where the guide's own example puts it, and in preAuthRef when the response has only one.

The validator is not part of make check, because it needs Java and a download of the guide and its dependencies. What it found is held by a test instead: each rule it enforced is asserted, so a change that reintroduces one of those errors fails.

The **Claims & auth** view converts a pasted 278 and shows the decisions, the notes and the resource.

8.3.6 Attachments, and why they need special handling

CMS-0053-F makes the 275 a HIPAA standard for claims attachments, with compliance required by 26 May 2028. It carries a document — an operative note, an imaging report, a scanned form — tied to the claim it supports.

The 277 is the other half of the same conversation. A payer receives a claim, decides it needs documentation, and sends a 277 asking for it; the provider answers with a 275.

A channel can route on the status code directly, because STC01 is a composite and its second component is the status. Written in canonical form, because a component is addressed with a dot and the two notations cannot be mixed — STC-1.2 is element 1, component 2:

```
filter: STC-1.2 in ("227", "233", "252", "287")
```

Those are the codes that unambiguously mean a document is wanted. The status **category** in STC-1.1 is deliberately not used on its own: R4 means a documentation request in some implementation guides

and "forwarded elsewhere" in others, so a filter on the category alone would put finalised claims in the queue for somebody to chase.

Two identifiers decide whether an attachment is ever associated with its claim, and both are reported: the **attachment control number**, which the provider put on the original claim to say documentation would follow, and the **trace number**, which matches the payer's request. An attachment arriving with neither is filed and never connected to anything, while the claim ages.

A document inside a 275 is **not interpreted**. Perfuse reports its bytes and what the sender says they are. An attachment may be a PDF, a TIFF, a CDA document or an HL7 message, and a channel needing the content can hand those bytes to the format that reads it.

The 006020 form, which is the one the rule adopts

CMS-0053-F adopts version 006020, and its 275 differs from the 005010 form in two ways that matter.

- **The payload is in BDS, not BIN.** BDS01 names a filter, ASC or B64. BDS02 is the length and BDS03 the data.
- **The payload is a MIME entity**, with headers saying what the document is and a base64 body. With B64 the entity itself is base64 as well.

Perfuse unwraps the entity and reports the document inside it, with its content type and file name. The bytes it reports are exactly the ones the sender wrapped. An entity that cannot be unwrapped leaves the raw payload in place, with a note saying why.

Building a 275. Perfuse also builds 006020X314 attachments: an unsolicited one to accompany a claim, or a response to a payer's 277 request. The difference is BGN01 02 against 11, and whether TRN02 carries the claim's PWK06 or the 277's TRN02. Every built interchange is parsed back before it is returned, and the response says whether the document inside is the one that went in.

The structure comes from the two published companion guides for 006020X314: the CMS esMD guide and UnitedHealthcare's. It does **not** come from the X12 technical report, which X12 licenses and which this project does not hold.

- Where the two guides agree, Perfuse follows them.
- Where they differ, it satisfies both if it can. CAT02 is HL for an HL7 document, which esMD uses, and IA for an image, which UnitedHealthcare uses.
- Otherwise it sends the optional segment only when given the value: esMD's REF*X1 claim identifier and UnitedHealthcare's DTP*472 service date.

There is no validation against the technical report. The partner's companion guide governs.

8.3.7 Electronic signatures on attachments

CMS-0053-F also adopts a signature standard, at 45 CFR 162.2002(e): the HL7 Implementation Guide for CDA Release 2, Digital Signatures and Delegation of Rights, Release 1. It applies to signatures put on attachment information when it is sent in a HIPAA attachment transaction, from 26 May 2028. A payer may not ask for more than the guide does.

Perfuse signs a C-CDA that way on its way into a 275, and checks the signatures on every C-CDA in a 275 it reads.

- **What is signed.** The whole document except its `legalAuthenticator` and `authenticator` participants, so each later signer leaves the earlier signatures intact. The signature says so with an XPath Filter 2.0 transform any XML signature library can evaluate.
- **Where it goes.** Base64 in `sdtc:signatureText` on the signing participant, with a one-line plain-text description. The signer is the document's legal authenticator if it has an unsigned one, otherwise a new authenticator.
- **What it states.** The signer's certificate, the time in UTC, a role from the NUCC provider taxonomy and a purpose from ASTM E1762. The purpose defaults to **Author's signature** for a legal authenticator and **Coauthor's signature** otherwise, as in the guide's own example.
- **How long it lasts.** The guide asks for XAdES-X-L, which carries everything a payer needs to verify the signature years later without reaching the network. That is the certification path, an OCSP response or CRL from the CA, and two time-stamps from a time-stamping authority. Perfuse fetches the revocation data from the addresses in the certificate and the time-stamps from `-tsa-url`.

If any of that is unavailable, the signature is still made and holds. The result names the XAdES form it reached, and says what stopped it short of X-L: for example a self-signed certificate (no CA to ask) or no time-stamping authority.

The key is `-signing-cert` and `-signing-key`. Put the CA certificates after the signer's in the same file so the path travels with the signature. For a signature that means what DSDR means, the certificate should name the clinician, with their NPI, rather than the server. Signing is an admin action, as signing any document is.

The guide does not name a namespace for its own elements. Perfuse puts `digitalSignature`, `authorizedSigner` and `SignaturePurpose` in `urn:h17-org:sdtc`. It also states the purpose as a XAdES commitment type with the ASTM OID, which XAdES tools understand. Delegation of rights, where a delegate signs with a SAML assertion from the authorised signer, is not produced, and a received one is reported as not verified.

The provider NPI is checked against its check digit. A payer matches an attachment to its claim partly on that number, and a mistyped one leaves the attachment unmatched while the claim is denied for missing documentation.

The **Claims & auth** view builds a 275 from a pasted C-CDA or an uploaded file, optionally signing the C-CDA, and reads one back with each signature's findings. Nothing on that page sends anything: the interchange is handed back to be delivered through a channel, which is where the partner, the credentials and the audit trail belong.

8.3.8 Binary segments

A BIN segment carries raw bytes, and raw bytes contain delimiters. Every other X12 segment ends at the segment terminator, so an ordinary parser finds segments by searching for one byte; an attachment cannot be found that way. A PDF containing a ~ would be cut into pieces, and each piece is a structurally valid segment — so the document is truncated, every segment after it is misaligned, and nothing reports an error.

Perfuse reads the byte count the segment declares and takes exactly that many bytes. Two consequences worth knowing:

- A declared length that does not match the data is **refused**, naming both numbers. The useful question is whether the sender's count was wrong or the file was truncated in transit, and only the two numbers answer it.
- BDS, the 006020 binary segment, keeps its count in BDS02 and its data in BDS03. It was refused until that could be confirmed from a published source rather than recalled, and two sources now confirm it: the CMS esMD companion guide's BDS table and UnitedHealthcare's guide.

The esMD guide's own sample interchange declares 16 bytes for the same 53-byte payload that its table correctly counts as 53. Perfuse refuses that sample, as it should. A test holds both the table's example and the sample's typo, so a change in either direction is noticed.

8.4 FHIR

8.4.1 What an HL7 v2 message becomes

The message type decides what is produced, and every conversion reports the judgements it needed as notes beside the bundle.

v2	FHIR
ADT	Patient, an Encounter when there is a visit, a Condition per DG1 diagnosis and an AllergyIntolerance per AL1. A28, A31, A29 and A24 are about the person, so they produce no Encounter or diagnoses, but do carry allergies.
ORU	DiagnosticReport per OBR, with its Observations and Specimen. Results are grouped under the OBR before them by position, because OBX-4 is not reliable.
ORM, OML, OMG	ServiceRequest.
SIU	Appointment, with the patient, the practitioners from AIP and the locations from AIL.
MDM	DocumentReference carrying the document.
VXU	Immunization, one per RXA.

Anything else produces the Patient and a warning naming the type that was not mapped.

Scheduling. A booking, a reschedule, a cancellation and a no-show all land on **the same Appointment**. Its identity comes from the appointment's own numbers in SCH-1 and SCH-2, never from the message, so a cancellation updates the booking. The alternative would be a second, cancelled appointment beside a live one, which shows the patient booked into a slot that no longer exists.

- **Status.** The filler status in SCH-25 decides it, and the trigger event is used only when that is empty.
- **Blocked slots.** A blocked slot is not an appointment with a patient, and produces none.
- **Start time.** It is read from AIS, AIG, AIL or AIP first, then SCH-11, because version 2.5 moved it and feeds did not all move with it.
- **Durations.** A duration in a unit Perfuse does not recognise is left out, not guessed.

Documents. Text notes (TX, FT, ST) keep their line breaks, including blank lines, because a blank repetition in a note is a paragraph break.

- **Embedded documents.** An embedded PDF, TIFF or JPEG (ED) is carried byte for byte. Base64 that is not valid base64 is refused rather than passed on.
- **Types.** The document type in TXA-2 is kept as sent and, where there is a clear equivalent, given the LOINC code US Core uses.
- **Signing.** A note nobody has signed is preliminary, and T11 makes the document entered-in-error.
- **No content.** A notification with no content (T01, T03 and the other odd-numbered events) produces no DocumentReference. FHIR requires content, and an empty attachment would look like the note while saying nothing.

This is what CMS Interoperability Framework criterion 14 asks for: chart notes and clinical documents returned as FHIR attachments.

Immunizations. A dose is never reported as given when the message says otherwise.

- **Refused doses.** A refused or not-administered dose (RXA-20 RE or NA) is not-done, with the reason from RXA-18.
- **Deletions.** A deleted one (RXA-21 D) is entered-in-error.
- **Historical records.** A record transcribed from somewhere else (RXA-9 other than 00) is not a primary source.
- **Unknown amounts.** An amount of 999 means unknown, not 999.
- **Routes and sites.** Each dose takes the route and site from the RXR after it, so a message with two doses does not give the second one the first one's route.

Awkward input. Each of these is reported in the notes rather than decided silently:

- **Identifiers.** PID-2 and PID-4, withdrawn after v2.7 but still sent, are kept like PID-3. CX.7 and CX.8 become

Identifier.period. An assigning authority carrying an ISO OID (AUSHIC&2.16.840.1.113883.3.879&ISO) becomes urn:oid:. A type code outside HL7 table 0203 is kept as text, not labelled as one.

- **Merges.** MRG on a merge event (A40 and its kin) sends the retired record too, inactive, with Patient.link both ways. On any

other event, such as an A08, nothing is merged and a warning says the prior identifier is still live.

- **Times.** A time of birth in PID-7 goes to the patient-birthTime extension. A timestamp with no offset takes the configured

timezone, otherwise the offset on MSH-7 (v2 presumes the sender's local time), otherwise UTC. A time of birth is the exception: with no offset from any of the three it is left out rather than labelled UTC, because a wrong offset can move a birth to another day. birthDate is kept, and a warning says the time was dropped and that setting a timezone for the feed keeps it.

- **Results.** Repeats of a text result are joined with line breaks. An ED result, or a coded one whose value is plainly Base64, becomes a

DocumentReference that the Observation points to in derivedFrom, since an R4 Observation has no attachment value. An SN comparator of = is dropped as redundant; <> keeps the text, because no quantity can say "not equal".

- **Visits.** An A08 with a discharge date (PV1-45) is a finished encounter. A missing or unknown patient class is the null

flavor UNK, because R4 requires a class.

- **Z segments** are named in a warning, since there is no standard map for them.

8.4.2 US Core 9.0.0 and USCDI v6

-us-core (and claim_us_core on a fhir destination) puts a US Core profile in meta.profile, but only on a resource that has been checked against that profile and found to conform. One that falls short is still converted and sent without the claim, and a note gives the reason, such as "US Core requires an encounter type, and this Encounter has none". The checks are transcribed from US Core 9.0.0, the version ONC's 2026 Standards Version Advancement Process approves for USCDI v6. They cover Patient, Encounter, the laboratory Observation and DiagnosticReport, Specimen, Practitioner, Organization, Location, Immunization, DocumentReference, the encounter-diagnosis Condition and AllergyIntolerance.

What the converter carries, by USCDI v6 data class:

USCDI v6	From	As
Patient Demographics	PID	Name, birth date, sex, address, phone and email, identifiers. Race and ethnicity (PID-10, PID-22) in US Core's extensions when coded in CDC Race and Ethnicity. Preferred language (PID-15) as BCP 47
Encounter Information	PV1	Encounter with type, class, period, location and participants
Facility Information	PV1-3	Location, with the facility as its managing Organization
Problems, Date of Onset	DG1	Condition (encounter diagnosis). DG1-5 is the onset, as the HL7 v2-to-FHIR map has it
Allergies and Intolerances, Reaction	AL1	AllergyIntolerance with category, criticality and reaction
Laboratory	ORU	DiagnosticReport, Observations, Specimen (from SPM)
Orders	ORM, OML	ServiceRequest
Immunizations	VXU	Immunization
Clinical Notes	MDM	DocumentReference
Care Team Members	PV1, TXA, DG1	Practitioner

Not carried, because v2 messages of these types do not hold them or Perfuse does not map them: Medications, Procedures, Goals and Preferences, Health Status Assessments, Care Plan, Family Health History, Medical Devices (UDI), Health Insurance Information, Vital Signs as a distinct profile, and Provenance.

8.4.3 R4 in both directions

A FHIR server declaring R4 accepts R4. That was not true of Encounters until this release. The server converted what it served to R4 but read what it received as if it were already in its internal form, so an R4 Encounter was refused with a 400. So was every bundle Perfuse's own ADT conversion produced, because each carries `Encounter.class`, which R4 writes as a single coding.

It was found the first time anything posted a converted ADT to Perfuse's own R4 server. A test now does that for every mapped message type, and another checks that everything written as R4 is read back unchanged.

8.4.4 Subscriptions

With `-fhir-subscriptions`, the FHIR endpoint sends notifications when an Encounter or an Appointment is created or changed. It follows the FHIR Subscriptions R5 Backport IG for R4, which is what CMS Interoperability Framework criterion 15 and TEFCA's FHIR roadmap use for appointment and encounter notifications.

A client creates a Subscription naming a topic, and optionally a filter such as `Encounter?patient=Patient/123`. The topics are listed in the capability statement:

Topic	Fires on	Filters
<code>https://perfuse.health/fhir/SubscriptionTopic/encounter</code>	Encounter created or updated	patient, status, class
<code>https://perfuse.health/fhir/SubscriptionTopic/appointment</code>	Appointment created or updated	patient, status

A filter that names anything else is refused, not ignored. An ignored filter is a subscription that notifies about every patient when the client asked about one.

The properties the implementation is built around:

- **An acknowledged change is never a lost notification.** The notification is recorded in the same database transaction as the change it describes, and delivered from there, so it survives a restart. A test writes an Encounter, starts a new server on the same database, and checks the notification arrives.
- **Notifications arrive in order, and gaps are visible.** Event 2 is not sent before event 1 is accepted, and each notification carries the running count, so a receiver can tell when it has missed one.
- **The server decides when a subscription is active.** A new subscription is requested, and becomes active only when the endpoint accepts a handshake. A client that asks for active gets requested. A refused handshake is tried again four times over about fifteen seconds before the subscription is an error: it goes out the moment the subscription is stored, often before the client that created it is listening.
- **A full-resource notification carries the version that changed.** A discharge notification carries the discharge, not whatever the Encounter had become by the time delivery succeeded.
- **Failure is bounded.** A refused delivery is retried with a growing delay. After ten consecutive failures the subscription goes to error: later changes are counted but not queued, and updating the subscription resumes delivery from the oldest held notification. The count then shows how many were missed.

What a client can make the server do is limited deliberately:

- Endpoints must be `https` unless the operator passes `-fhir-subscriptions-allow-http`, which is for a test receiver.

- The egress policy is applied when a subscription is created and again at every delivery, and redirects are not followed.
- The payload defaults to `id-only`, which says what changed and lets the receiver fetch it with its own credentials.
- Nothing a receiver sends back is shown to anybody, only its status code.

It is off unless asked for, because once on, a FHIR client can make this server send patient data to a URL of the client's choosing. It needs an R4 endpoint that accepts writes, and refuses to start otherwise.

Not implemented, so nobody has to find out by trying:

- channels other than `rest-hook`
- heartbeats
- notifications on delete
- the `$status` and `$events` operations

The **Subscriptions** view shows each subscription's topic, filter and endpoint, how many events it has had, how many were accepted and how many are waiting, and the last reason one was refused. Header values and the endpoint's query string are withheld, because that is where a subscriber's credentials live.

8.5 NCPDP Telecommunication Standard

Pharmacy claims. Fixed-width header followed by variable segments.

Two details that matter, both of which are easy to get wrong and produce plausible results:

The header is exactly 56 bytes and fixed width. A short header cannot be detected by length alone, because a truncated header plus the following segments still exceeds 56 bytes. Detection is that a correct header never contains a separator byte, so the header is scanned for one and a message that has one is refused with the number of bytes it is short by.

`0x1E` and `0x1C` are group and field **start** markers, not separators. Treating them as separators shifts every field one place left, and most of the resulting values stay plausible — a quantity lands where a day supply was expected and both are numbers.

NCPDP and SCRIPT channels are excluded from MLLP framing, and this is enforced rather than advised. A claim legitimately contains `0x1C`, which is MLLP's own end-of-block byte, so framing a claim would truncate it at the first field marker.

8.6 NCPDP SCRIPT

Electronic prescriptions, in XML. The internal package is called `eprescribe` because `script` was taken by the JavaScript engine.

The central safety concern is the substitution flag, and it reads backwards from the intuition: on the wire, 0 means substitution is **allowed** and 1 means dispense as written.

An absent substitution flag is refused rather than defaulted. Both possible defaults reach the patient — one may dispense a generic where the prescriber required the brand, the other prevents a substitution the prescriber permitted — so there is no safe assumption and the message stops.

A Schedule II prescription with refills is refused, and the builder will not produce one.

Refusals and warnings are kept as separate lists rather than one severity-tagged list, because a reader who has to check a severity field to know whether the message was sent will eventually not check it.

8.7 DICOM

Imaging. Perfuse can receive DICOM objects, query a remote archive, and store to one.

DICOM is not a message format in the same sense as the others — objects are large, and the association handshake matters more than the payload parsing. The `called_ae` and `calling_ae` titles are the commonest reason a first connection fails, and the failure looks like a network problem rather than a configuration one.

8.7.1 What a C-STORE listener accepts

Left open, a DICOM listener accepts anything from anyone that can reach the port. Four settings under **What this listener accepts** narrow that, and the first is the one worth understanding.

Only accept these calling AE titles is the restriction. The `*called*` AE title — the one alongside the address — is the name a sender dials, so it identifies this listener rather than the sender, and it is not a restriction. Anybody who can reach the port can use the correct called AE title, because it is published to them. Without a list of permitted callers, any host on the network may push images into a clinical channel.

Only accept these SOP classes and **only accept these transfer syntaxes** take comma-separated UIDs. Empty accepts every kind of object and negotiates whatever compression the sender offers, which includes kinds this channel's destinations cannot handle. Refusing at the association is clearer to the sender than accepting the object and failing afterwards.

Largest object accepted is in bytes, and zero means no limit. Whole-slide images and long series run to gigabytes, so the limit is otherwise memory.

8.8 Delimited

CSV and its relatives. The options are on the source in the builder, and the format reference lists the file keys.

Paths address columns. A named column is available when a header row is declared or the names are given; otherwise columns are positional.

8.8.1 Naming the columns

There are two ways, and they are exclusive. Either the first row of the file names them — **the first row names the columns** — or you list them yourself, in order. Declaring a header means the row is not treated as data.

If neither is set, columns can only be addressed by position, so a filter written against a name has nothing to match. Turning the header on removes the list of names rather than keeping both: with a header the names come from the file, so a list left behind would be ignored, and a channel that reads as though it chose the names when the file chose them is harder to debug than one that refuses.

8.8.2 What a filter or a script sees

One delimited document becomes many messages, so this needs stating before you write either.

With splitting on, everything runs per row. Declarative steps, the filter and the transformer script all see one row, and a path means **this** row's column. A filter that rejects a row rejects that row and no other, and the received count is rows rather than documents.

With splitting off, the message is the whole document and a write that crosses rows is refused rather than applied to the first one. That refusal is deliberate: silently writing to row one is the behaviour that produces a file which looks transformed and is not.

8.8.3 The three that change what counts as an error

Accept rows with the wrong number of columns is off by default. A row with too many or too few columns is then an error, which is what catches a file whose shape changed without warning. Turning it on accepts the row and leaves the missing columns empty.

Trim spaces around each value fixes the common case of a value padded to a column width. Leave it off where trailing spaces are significant, which they occasionally are in an identifier.

Keep blank lines as rows is off by default and usually should be. On, a blank line becomes a row of empty columns, which matters only when a count has to match the file exactly.

8.9 Raw

No parsing at all. The message is bytes.

Use this when the content is not something Perfuse understands and does not need to: a PDF being moved between systems, a proprietary format, an already-formed document being forwarded unchanged.

The cost is that nothing which depends on structure works — no field-level transformation, no filter on a field, no field-level comparison in the parity checker. A raw channel is a pipe, and it is honest about being one.

8.10 Choosing

If the message is HL7 v2, use `hl7`, even if you only intend to forward it unchanged. Parsing gives you the trace, the field-level statistics, the ability to filter later and the ability to compare. `raw` on parseable content buys nothing and loses all of that.

Use `raw` when parsing would fail or would be a lie about the content.

8.11 SMART Health Links and Cards

CMS's **Kill the Clipboard** asks providers to accept a patient's records from a QR code at check-in, and to hand the visit record back the same way. The QR code holds a **SMART Health Link** (`shlink:/...`): an address, a 256-bit key and, optionally, a label, an expiry and a passcode flag. **Exchange → Health links** reads one.

Receiving. Paste the link, or scan it with the camera where the browser can (Chrome and Edge).

Perfuse fetches the manifest, decrypts each file with the key from the link, summarises any FHIR bundle - who the patient is, how many of each resource - and checks any **SMART Health Card** inside against its issuer's published key. **Deliver to the channel** hands the FHIR content to a channel as a message, which is how it reaches the EHR. The fetch is held to the egress policy at connect time, plain HTTP is refused, and the response of a server that is not a manifest server is never shown, because the address came from a stranger's QR code.

Sharing. **Share** encrypts a FHIR bundle, hosts it, and shows the link and its QR code once. The server keeps the ciphertext and not the key - it cannot open what it hosts, and cannot show the link again. A passcode is hashed; ten wrong ones disable the link. Links expire after 30 days by default, can be revoked, and list how often and by whom they were fetched. Start the server with `-public-url` set to the https address patients' phones reach; the manifest URL it builds must fit the specification's 128 characters.

Verifying a card. **Verify a card** takes a numeric `shc:/` QR (one chunk or several), a `.smart-health-card` file or a JWS, and says whether the signature matches the issuer's key.

9 Acknowledgements

An acknowledgement tells the sender what happened. For HL7 v2 over MLLP it is an ACK message; the sender will not send the next message until it arrives, and will usually resend if it does not.

Getting acknowledgements wrong is the most common way an integration silently loses data, because a sender that is being acknowledged has no reason to complain.

9.1 Timing

Covered in message flow, and the summary is: `ack.when` is `on_delivery` by default, meaning the acknowledgement is sent only after every enabled destination accepted the message. `on_receipt` sends it as soon as the message is parsed and queued.

9.2 What Perfuse sends

An AA when the message was accepted, an AE when it was rejected for a reason the sender could act on, and an AR when it was rejected for a reason the sender cannot.

The distinction between AE and AR is who has to do something. AE means the message had a problem — a field that would not parse as a date, a required segment missing — and the sender can correct it and resend. AR means Perfuse could not accept it regardless of content, which usually means a configuration or availability problem at this end. A sender that resends after an AR will get the same answer.

9.3 Identifying this engine

`ack.application` and `ack.facility` populate MSH-3 and MSH-4 of the acknowledgement.

When both are empty the acknowledgement mirrors the original receiver — it uses whatever the incoming message had in MSH-5 and MSH-6. That is what most senders expect, and it is the default for that reason.

Set them explicitly when the sender validates them, or when several Perfuse instances handle the same feed and you need to know which one answered.

9.4 The trigger event

`ack.include_trigger_event` sends `ACK^A01` rather than a bare ACK.

Some receivers require it and others reject it, which is why it is an explicit option with no clever default. If a sender is refusing your acknowledgements and the content looks right, this is the first thing to try.

9.5 Negative acknowledgements are not failures to hide

A NACK treated as a success is one of the most damaging misconfigurations possible, and it is common enough to be worth naming. It happens when a destination sends back a negative acknowledgement and the sending side records the **transmission** as successful because the bytes went out and something came back.

Perfuse reads the acknowledgement it receives from an MLLP destination and records the outcome accordingly. An AE or AR from a downstream system is a delivery failure, is retried according to the destination's policy, and counts towards the failure rate that alerts fire on.

If you are migrating from another engine, this is worth verifying explicitly on your own traffic rather than assuming. The parity checker will show it: a destination that the old engine recorded as delivered and Perfuse records as failed usually means the old engine was ignoring a NACK.

9.6 Timeouts

Waiting for an acknowledgement is bounded. When it does not arrive in time the delivery is a failure and is queued for retry.

A timeout is not the same as a rejection and is recorded differently. A rejection means the far side considered the message and declined it; a timeout means you do not know whether it was processed. That distinction decides whether a retry is safe: retrying after a rejection is pointless, and retrying after a timeout risks a duplicate. Perfuse retries after a timeout, because a missing clinical message is generally worse than a duplicate one, and the receiving system is generally able to detect a repeated message control ID.

Whether that trade is right for a given feed is a judgement, and it is worth making deliberately for anything where a duplicate has a clinical consequence — an order, a medication administration — rather than accepting the default silently.

9.7 Sources that do not acknowledge

Not every transport has the concept. A file source has nowhere to send an acknowledgement; a broker source acknowledges to the broker rather than to the original sender; an HTTP source answers with a status code.

Where the source cannot deliver the acknowledgement mode a channel asks for, the channel is refused at load rather than starting and silently not acknowledging. A channel that believes it is acknowledging and is not is worse than one that does not start.

9.8 X12 acknowledgements

Everything above is about HL7 v2. X12 has its own acknowledgements, and unlike v2 they are **off by default**: an empty setting means none.

That default is the safe one, because a trading partner who is not expecting an acknowledgement and receives one may treat it as an unsolicited interchange. The acknowledge key on an X12 channel takes one of four values:

Value	What is sent
none	Nothing. The default.
999	Implementation acknowledgement. Supersedes the 997 for HIPAA transactions.
997	Functional acknowledgement. Older, and still what many payer connections expect.
ta1	Interchange acknowledgement, about the envelope rather than its contents.

Which one to send is a property of the trading partner relationship rather than of the message, which is why it has to be configured and cannot be inferred. A 999 supersedes a 997 for HIPAA transactions, but plenty of partners — older payer connections especially — are set up to expect a 997 and will treat a 999 as an unrecognised file.

Sending the wrong acknowledgement is worse than sending none. It answers a question nobody asked while leaving the real one open, and the sender has no way to tell the difference.

9.9 HL7 v3 acknowledgements

An HL7 v3 channel sends an acknowledgement by default, which is the opposite of X12 and worth knowing.

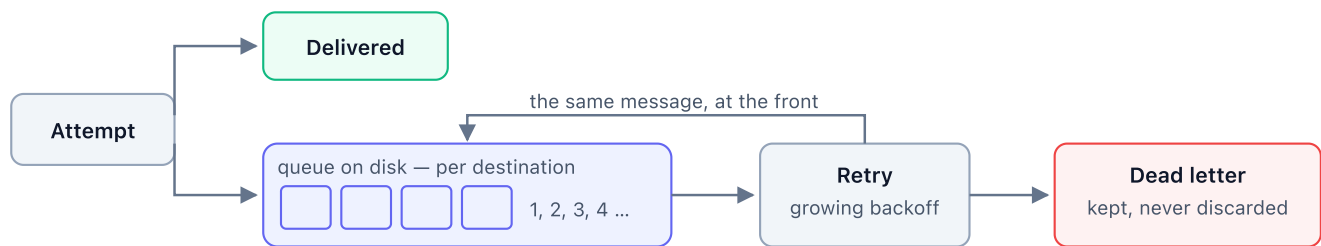
The reason is the transport rather than the standard: a v3 interaction generally arrives over a synchronous connection with the sending application waiting on a reply, and a sender that receives nothing will usually retry. That is how a patient gets registered three times.

Turning it off is possible and has to be done explicitly — "not set" and "set to false" are deliberately different, so the default stands unless somebody said otherwise.

10 Delivery, Queueing and Retry

A destination that cannot be reached is the normal case, not the exception. Systems restart, networks drop, and a hospital interface engine is unavailable during its own maintenance window. Delivery is built around that.

10.1 Queue per destination



Each destination has its own queue, so one slow receiver does not hold up the others.

The queue is on disk, so a restart does not lose it. Order is preserved: message 2 is not attempted before message 1 succeeds or is given up on. A message that exhausts its attempts is dead-lettered and kept. Nothing is deleted to make a queue look healthy.

Each destination has its own queue. A destination that is failing does not affect the others, and each has its own depth, age and retry state.

Queueing is per destination rather than per channel because the failure is per destination. A channel-wide queue would hold up delivery to three working systems because a fourth is down, and the two situations — one destination unavailable, all four unavailable — would look the same in the statistics.

10.2 Queue settings

queue on a destination:

Key	What it does
<code>enabled</code>	Whether failed messages are queued at all.
<code>max_attempts</code>	How many times to try before giving up.
<code>backoff</code>	Wait before the first retry.
<code>max_backoff</code>	Ceiling on the growing wait.
<code>max_depth</code>	Most messages to hold. Unset means no limit.
<code>retain_hours</code>	How long a delivered message is kept. Unset keeps it indefinitely.

The last two were missing from this table and from the builder, and they are the pair that stops a disk filling. A queue with neither grows until the filesystem is full, and that failure arrives as something unrelated breaking — a write failing somewhere else entirely — rather than as a queue problem.

Backoff grows between attempts up to `max_backoff`. Growing rather than fixed, because a destination that is down is usually down for minutes rather than seconds, and retrying every second for ten minutes produces several hundred log entries that bury the one that matters.

10.3 Ordering is preserved

Within a destination, messages are delivered in arrival order, and a message that fails holds its place. Later messages wait behind it rather than overtaking.

This is the right default for clinical data — an admission and a discharge applied in the wrong order is worse than both being late — and it has a consequence worth understanding: **one message that always fails will hold up everything behind it.**

That is why queue depth and queue age are separate alert conditions. A deep queue that is draining is a busy system. A queue whose oldest message keeps getting older is stuck, and the alert kinds `queue-depth`, `queue-age` and `queue-stuck` distinguish them.

A poison message — one that will never succeed no matter how often it is retried — is the specific failure this design is vulnerable to. `max_attempts` is what limits the damage: after it, the message is recorded as failed and the queue moves on. Setting `max_attempts` very high in the hope of eventual success converts a single failed message into a stopped feed.

10.4 When queueing is disabled

With `queue.enabled` false, a failed delivery is a failed delivery. It is recorded and not retried.

That is the right choice when late data is worse than no data — a real-time display, a paging system, anything where a message delivered forty minutes after the event would be misleading rather than merely delayed.

It is the wrong choice for anything that goes into a record, where the message is just as valid an hour later.

10.5 Retry versus queue

There are two mechanisms and they operate at different levels.

retry on a destination governs immediate attempts within one delivery — a connection that fails is tried again after a short pause, without leaving the delivery path. This handles transient faults: a dropped connection, a momentary refusal.

queue governs what happens once those immediate attempts are exhausted. The message leaves the delivery path and is retried later on a longer schedule.

Both exist because the two failures are different. A connection reset on the first attempt is usually gone by the second, and going through the queue for it would add minutes of latency to a fault that lasted milliseconds. A destination that has been switched off for maintenance will not come back within three immediate attempts, and hammering it is pointless.

10.6 What an HTTP endpoint counts as delivered

An HTTP destination has three settings that decide whether a message is treated as sent, and all three are on the destination in the builder.

Status codes that mean delivered. Empty means any 2xx. Some endpoints answer 202 Accepted for something they have not processed yet, and some answer 200 with an error in the body — the second is what `*Treat as failed if the reply contains*` is for. Setting an explicit list is worth doing when a partner has told you which code means what, because the alternative is discovering the difference from a message that was never delivered and never queued.

Follow redirects. Off by default, and deliberately. A redirect can point at a different host, so following one sends the message somewhere nobody configured and reports success.

Bearer token. Sent as an `Authorization` header. It is stored with the channel and never shown again, so leaving the box empty when editing an existing channel keeps the token already saved rather than clearing it.

10.7 Naming and pruning an archive folder

A file destination writes one file per message by default, named with a timestamp and the message's control ID. Three settings on the destination change that.

File name takes `${...}` placeholders, so a name can carry the date or the message type. This is worth setting: the default is correct and unfindable, and a folder of timestamps is one nobody searches.

Suffix while writing defaults to `.part` and is removed by a rename once the file is complete. Whoever collects these files should not pick up a half-written one, and a rename within a folder is atomic. Change it only if the collector filters on a different extension.

Delete after prunes files older than the given number of hours. Empty or zero keeps them forever, which is the default and is a decision rather than an oversight — but an archive nobody prunes fills the disk, and a full disk stops the channel rather than the archiving.

10.8 Sending to a FHIR server

Three settings decide how much is checked before a bundle leaves.

Claim US Core conformance adds the profile to each resource. Only claim it when the resource carries what the profile requires, because a receiver may validate against the claim and reject a bundle that would otherwise have been accepted.

Validate before sending checks each resource here rather than learning from the receiver. A rejection at the far end tells you less, later, and often only in a log you cannot read.

Treat warnings as rejections only has an effect when validation is on, and the control is unavailable until it is. It is strict enough to stop a bundle a receiver would accept, so it belongs on a feed being brought up rather than one in service.

10.8.1 Hosted FHIR services

A `fhir` destination's `auth` block signs in to the hosted services, where a pasted bearer token stops working within the hour:

```
fhir:
  url: https://healthlake.us-east-1.amazonaws.com/datastore/<id>/r4/
  auth: {type: aws, access_key_id: ${AWS_ACCESS_KEY_ID}, secret_access_key:
    ${AWS_SECRET_ACCESS_KEY}}
```

`aws` signs each request for AWS HealthLake with Signature Version 4, with the region read from the URL. `azure` gets a Microsoft Entra ID token for Azure Health Data Services from `tenant_id`, `client_id` and `client_secret` - an app registration holding the FHIR Data Contributor role - with the FHIR service URL as its audience. `client_credentials` does the same against any OAuth 2.0 `token_url`. Tokens are reused until a minute before they expire. The same choices are under the destination's **Hosted service sign-in** in the channel builder.

10.8.2 Public health case reporting

Tick **Report reportable conditions to public health** on a FHIR destination and it sends case reports rather than bundles. Each message is checked for a reportable-condition trigger code: a diagnosis, a lab test or a coded result. If it has one, the message becomes an HL7 eCR 2.1.2 eICR, wrapped in an eCR message and posted to the URL's `$process-message`. If it has none, nothing is sent and nothing fails; the destination counts it as not reportable.

```
fhir:
  url: https://ecr.agency.example/fhir
  ecr:
    rctc: /etc/perfuse/rctc.json
    source: https://fhir.springfield-general.example/fhir
    facility: {name: Springfield General Hospital, phone: +1-217-555-0100, city:
Springfield, state: IL}
```

rctc is the Reportable Conditions Trigger Codes, as a FHIR ValueSet or the eRSD Bundle. It needs a UMLS licence, so it is not included; without it a short built-in sample is used, for testing only. source is where the agency sends its Reportability Response. The facility is required because eCR asks for its phone and address, and a v2 message has neither.

A report says what the message lacked rather than inventing it, and each gap is logged. Try a message in the FHIR lab's **Public health case report** panel before pointing a live feed at an agency.

10.8.3 Lab results to public health: ELR

perfuse elr -config elr.yaml results.hl7 turns a lab's results into the HL7 2.5.1 ELR messages state health departments receive, one for each message with a reportable result, and only the orders that carry one: a complete blood count beside a positive SARS-CoV-2 test is not public health's business. The trigger codes are the same as case reporting's, the RCTC you load or the built-in sample.

ELR names everybody by OID or CLIA number, which no lab feed carries for the sender and the receiver, so those come from the file:

```
sending_application: {namespace: Perfuse, id: 2.16.840.1.113883.19.5.9, type:
ISO}
sending_facility: {namespace: Springfield Lab, id: 2.16.840.1.113883.19.5, type:
ISO}
receiving_application: {namespace: ELR, id: 2.16.840.1.113883.19.6.1, type: ISO}
receiving_facility: {namespace: State DOH, id: 2.16.840.1.113883.19.6, type: ISO}
processing: T
ordering_facility: {name: Springfield Clinic, phone: "+1 217 555 0100", line: 1
Main St, city: Springfield, state: IL, postalCode: "62701"}
performing_lab: {name: Springfield Lab, line: 2 Lab Rd, city: Springfield, state:
IL, postalCode: "62701"}
performing_lab_clia: 14D0000000
placer_authority: {namespace: SPRINGFIELD-EHR, id: 2.16.840.1.113883.19.5.1,
type: ISO}
filler_authority: {namespace: SPRINGFIELD-LAB, id: 2.16.840.1.113883.19.5.2,
type: ISO}
```

The ordering facility, performing lab and assigning authorities are used only where the lab's message leaves them out, and each use is a note. What only the lab knows is never filled in: a result with no

specimen type goes with SPM-4 empty and a note saying so, because a guessed specimen in a report to public health is worse than a gap the state can ask about. Start with processing: T while the state's onboarding team tests the feed.

What the lab got right is kept as it was sent. A child order (a susceptibility panel or a reflex test whose OBR-29 names its parent and which has no ORC of its own) goes as received, with no ORC or specimen invented for it, and a child with no reportable result of its own goes with its parent's, since a susceptibility means nothing without the culture it belongs to. A note after the PID stays with the patient, and ORC-14, the call-back phone number, is copied from OBR-17 when the lab only put it there. All 28 example messages ONC publishes for its ELR certification tool come out valid with no errors.

MSH-2 is written as ^~\&#, the five characters ELR Release 1's errata require and ONC's certification tool checks. If your state's guide shows the older four, add `encoding_characters: "^~\&"` to the file.

To send them from a channel rather than from the command line, give the destination that reaches the state an `elr` block naming the same file. Each lab result is reshaped on the way out, and one with nothing reportable is not sent at all, which is counted as delivered, not failed: the state asked for reportable results only.

```
destinations:
  - name: state-elr
    type: mllp                # or http, sftp, file: whatever the state's
onboarding team gives you
    address: elr.health.state.example:6661
    tls: {enabled: true}
    elr:
      config: elr.yaml        # beside this channel file
      rctc: rctc.json         # optional; the built-in sample is for testing
only
```

The block applies on an HL7 v2 channel, to destinations that deliver the message itself (mllp, http, tcp, file, sftp, ftp, S3, Azure Blob, soap, the message brokers, or another channel); the channel refuses to load otherwise, and when the file is missing or lacks the sending facility, receiving application or receiving facility. The notes are written to the log with the message's control ID.

The ELR file is the channel's companion, not a channel: `perfuse check` and the Channels screen skip it rather than reporting a channel that will not load. In the channel builder the same two settings are under the destination, in **Lab reporting (ELR 2.5.1)**; the trigger codes field opens once an ELR file is named, and saving refuses a file that is not beside the channel.

10.9 When the transport says yes and the message did not arrive

"Did this arrive" is often not a question the transport can answer. An MLLP receiver returns an application acknowledgement whose meaning is in its text. An HTTP receiver returns 200 with an error document. In both cases the transport succeeded and the message did not arrive.

Inspect the reply on a destination takes a script given the receiver's response. Return false, or throw, and the delivery is marked failed — which means it retries, queues, and appears in the failure count instead of being silently counted as delivered.

It is offered only on destinations that receive a reply. On one that cannot, it is refused at load rather than left never running.

10.10 What a delivery outcome means

Outcome	Meaning
Delivered	The destination accepted the message. For MLLP, a positive acknowledgement was received.
Queued	Delivery failed and the message will be retried.
Failed	Delivery failed and will not be retried, either because queueing is off or attempts are exhausted.
Filtered	The destination declined the message by its own filter. Not an error.

These are recorded per destination per message, so a message that reached three of four destinations is recorded as exactly that. There is no channel-level "success", because it would not mean anything.

10.11 Slow delivery

A destination that is accepting messages but taking a long time about it is a distinct problem from one that is failing, and it is easy to miss because every outcome is a success.

The `slow-delivery` alert kind exists for this. It is worth setting on anything where the sender's acknowledgement waits on delivery, because a destination whose response time has quietly doubled will eventually cross the sender's timeout and start producing resends — and at that point the symptom appears upstream, in a system you may not administer.

10.12 Writing to a database

`type: database` runs a statement per message with values bound from the message.

Placeholders are the driver's own — `$1` for postgres, `?` for mysql and sqlite, `@p1` for sqlserver — and Perfuse does not rewrite them. That is deliberate: a translation layer that got a placeholder wrong would bind a patient's name to the wrong column, and the result would look like valid data rather than an error.

A statement written in the wrong dialect is refused when it is saved, naming the style the driver wants. Before that check existed the mistake surfaced as `syntax error at or near ",,"` from the server when a message arrived, which names a statement you did write and so sends you looking for a typo instead of at the dialect.

```
destinations:
  - name: warehouse
    type: database
    database:
      driver: postgres
      dsn: ${WAREHOUSE_DSN}
      statement: INSERT INTO messages (mrn, family_name, birth_date) VALUES ($1,
$2, $3)
    params:
      - PID-3.1
      - PID-5.1
      - PID-7
```

A quoted parameter is a constant rather than a path, which is how a channel writes its own name or a source system code into a row beside the message data.

Values are always bound as parameters. A `${...}` reference inside the statement is refused: a name containing an apostrophe would break the statement, and a hostile value would rewrite it.

10.13 Sending to a raw socket

`type: tcp` writes to a socket with framing you choose, which is what a device or a laboratory instrument usually wants. MLLP is one framing among several; this destination offers the others.

```
destinations:
  - name: instrument
    type: tcp
    tcp:
      address: instrument.lab:9100
      framing: delimited
      delimiter: \r
      expect_reply: true
      timeout: 30s
```

Framing has no default, deliberately. Writing with the wrong framing does not fail. The far end reads messages split in the wrong places, or waits for a terminator that never comes, and the symptom appears at the receiving system rather than here.

framing	What it writes
mlp	0x0b before, 0x1c 0x0d after. The same framing an MLLP destination uses.
delimited	A delimiter after each message, and an optional start_block before it.
fixed	Records padded or truncated to record_length.
length	A length_bytes prefix. big_endian and length_includes_header both have two conventions in the wild and the difference is silent.
whole	One message per connection, closed to mark the end.

Only the settings the chosen framing uses are accepted. A record length on a delimited stream is refused rather than ignored, because a setting that is silently ignored is one somebody believes is in effect.

expect_reply changes what delivered means. Without it, success means the bytes reached the operating system's send buffer — which a peer that crashed a moment later never read.

10.14 Amazon S3, SQS and SNS

A channel can read from an **SQS queue** or an **S3 bucket prefix**, and deliver to SQS, SNS or S3. All five are signed HTTP with no AWS SDK in the binary, and all five take the same access settings: region, access_key_id, secret_access_key (write \${AWS_SECRET_ACCESS_KEY} to read the environment), an optional session_token, and endpoint for LocalStack, MinIO or a VPC endpoint. Under **Channels → New channel**, choose **We read from an Amazon SQS queue**, **We collect objects from an Amazon S3 bucket**, or the SQS, SNS and S3 destination types.

Nothing is acknowledged to AWS until the channel has handled it. An SQS message is deleted afterwards and otherwise reappears when its visibility timeout ends, so a queue's redrive policy is what moves one that keeps failing to a dead-letter queue. An S3 object moves to processed/ afterwards, or to error/ if it could not be handled, so one bad file does not stop the rest.

On a FIFO queue or topic, messages are grouped by patient by default - SQS keeps order within a group and nowhere else - and deduplicated by content, so a retry is not delivered twice.

An S3 archive can be written straight to **Glacier** with storage_class, and as **JSON lines for Amazon Athena** with format: ndjson; perfuse athena prints the table. The reference has every setting.

10.15 AMQP 1.0 and Azure

AMQP 1.0 is what Azure Service Bus and Event Hubs speak, and RabbitMQ 4, ActiveMQ and Artemis beside their other protocols. An amqp source receives from a queue or a subscription and accepts each message only once the channel has handled it; one that could not be handled is released as a failed delivery, so the broker redelivers it and, past the queue's limit, dead-letters it. An amqp destination

reports success only when the broker accepts the message. For Service Bus: `addr` is `<namespace>.servicebus.windows.net:5671` with `tls`, username the shared access policy name, password its key, and address the queue name, or `<topic>/subscriptions/<name>` to receive.

Azure Blob Storage: an `azure_blob` destination writes a blob per message in the access tier chosen - Hot, Cool, Cold or Archive - and an `azure_blob` source picks blobs up from a prefix, moving them to `processed/` or `error/` as the S3 source does. Prefer a SAS token limited to the container over the account key.

10.16 MQTT

An `mqtt` source subscribes to a topic filter (+ matches one level, # every level below) on any MQTT 3.1.1 broker: Mosquitto, EMQX, HiveMQ, AWS IoT Core, Azure Event Grid. At QoS 1, the default, each message is acknowledged only once the channel has handled it. MQTT has no way to refuse a single message, so one that could not be handled is left unacknowledged and the connection is made again; the broker keeps the session under the client id (`perfuse-<channel>` unless `client_id` says otherwise) and sends it again, first in line. QoS 0 is at most once: anything sent while Perfuse is down is lost. An `mqtt` destination publishes and, at QoS 1, reports success when the broker has the message.

```
source:
  type: mqtt
  mqtt:
    addr: broker.hospital.local:8883
    topic: hl7/+/adt
    username: perfuse
    password: ${MQTT_PASSWORD}
    tls: {enabled: true, ca_file: /etc/perfuse/broker-ca.pem}
```

AWS IoT Core wants a client certificate: give `tls.cert_file` and `tls.key_file`.

10.17 IBM MQ

`ibm_mq` reads and writes IBM MQ queues through the queue manager's messaging REST API (`mqweb`), which IBM MQ 9.1.5 and later ship on every platform, z/OS included, so no MQ client is installed beside Perfuse. The user needs the `MQWebUser` role and put, get and browse authority on the queue. A source browses the first message, hands it to the channel, and removes it by its message id only once it was handled; a message that fails stays first on the queue. Run one reader per queue. A destination puts persistent messages (unless `non_persistent`), with the HL7 control id as the correlation id.

```

destinations:
- name: to-mainframe
  type: ibm_mq
  ibm_mq:
    url: https://mq.hospital.local:9443
    queue_manager: QM1
    queue: HL7.ADT.OUT
    username: perfuse
    password: ${IBM_MQ_PASSWORD}
    tls: {enabled: true, ca_file: /etc/perfuse/mq-ca.pem}

```

`api_version` is v3 for MQ 9.3 and later (the default), v2 for 9.1.5 to 9.2. A queue manager with an AMQP channel can also be reached with the `amqp` connector.

10.18 Google Cloud Pub/Sub and Cloud Storage

A `pubsub` source pulls from a subscription and acknowledges each message once handled; one that fails is handed back at once (its acknowledgement deadline set to zero), with the rest of its batch, so Pub/Sub delivers it again and, past the subscription's dead-letter policy, dead-letters it. A `pubsub` destination publishes with the HL7 message type and control id as attributes; `order_by: patient` sets the ordering key from PID-3.1, so a subscription with message ordering keeps each patient's messages in order.

A `gcs` destination writes an object per message, named as an S3 key is, in the storage class chosen (STANDARD, NEARLINE, COLDLINE, ARCHIVE). A `gcs` source picks objects up from a prefix and moves them to `processed/` or `error/` as the S3 source does.

Both sign in as a service account: `credentials_file` is its JSON key. Leave it empty on Google Cloud (a VM, GKE or Cloud Run) to use the service account attached there. `endpoint` names an emulator (the Pub/Sub emulator, `fake-gcs-server`), which takes no credentials.

```

source:
  type: pubsub
  pubsub:
    project: hospital-integration
    subscription: adt-to-perfuse
    credentials_file: /etc/perfuse/gcp-key.json

```

10.19 Delivery to another channel

A channel destination hands the message to another channel by name. The receiving channel treats it like any other arrival: it records it, filters it, transforms it and fans it out.

This is the supported way to build a chain. It is deliberately visible — two channels, two sets of statistics, two entries in the message history — because a chain hidden inside one channel is a chain nobody can monitor.

The message is handed over in memory rather than over a network, so there is no framing, no acknowledgement and no timeout between the two.

10.20 What happens when the process is killed

The question anyone who has run an interface engine asks first, and the one that is rarely answered, because the honest answer is usually embarrassing.

It was answered by killing the process with SIGKILL part way through a batch — which is what a power failure, an out-of-memory kill and a hypervisor reset all look like from inside — and counting what survived against what had been promised.

The property under test is not "nothing is lost". That is not achievable and claiming it would be a lie. It is the narrower promise a sender actually relies on: **a message that was positively acknowledged is a message that arrived.**

With `ack.when: on_delivery`, which is the default, every destination is written before the acknowledgement goes out, so an AA is a statement about the destination rather than about a queue. A message killed before its acknowledgement may well be lost, and that is correct: the sender was never promised anything and will send it again.

10.20.1 Acknowledging on delivery

Five runs, killed after 5, 25, 40, 75 and 110 messages of 120.

Acknowledged	Present downstream afterwards	Acknowledged but missing
255	255	0

The promise held at every kill point, and nothing was duplicated. A duplicate would have been acceptable and a loss would not: a resent A08 is a nuisance receivers absorb, and a lab result that silently never arrived is a patient safety event.

10.20.2 Acknowledging on receipt

`on_receipt` answers as soon as the message is queued, before any destination has been written. This chapter has always said an acknowledged message can still be lost that way. Here is the size of it.

Killed after	Acknowledged	Present downstream	Lost
5	5	1	4
25	25	1	24
75	75	2	73
110	110	2	108

Essentially everything in flight. The setting is not a defect and there are feeds where it is the right choice, but it should be chosen knowing that a crash discards what has been acknowledged and not yet written, and that this is almost all of a burst.

Nothing recovered on restart, because `perfuse run` has no durable store. A destination that must not lose messages during an outage needs `queue.enabled`, which needs the database that `perfuse serve` provides — and `perfuse run` now refuses such a channel rather than starting it without the queue.

10.20.3 A sender that dies half way through a message

The most dangerous of these faults, because a truncated HL7 message usually still parses. The segments before the cut are complete and well formed, so a receiver has no way to know that PID and PV1 arrived and the OBX segments carrying the results did not.

A frame was opened, six tenths of a message sent, and the connection reset rather than closed. The partial message was not delivered, the listener survived, and a well-formed message sent immediately afterwards was accepted normally, so the fragment was not carried into the next message either.

The connection log now reports how many bytes of an unfinished frame were abandoned, for every failure mode rather than only a tidy close. Previously a crashed sender, a killed process and a pulled cable all logged `messages=0` and nothing else, which cannot distinguish an empty health check from a lab result thrown away nine tenths of the way through.

10.20.4 A full disk

Space exhaustion is detected and logged rather than swallowed. Two things about it are worth knowing.

The sender does not get a prompt rejection. The delivery retries on the usual schedule — one second, two, four, eight — and the negative acknowledgement arrives about fifteen seconds later. A sender whose own timeout is shorter than that sees a stalled connection rather than an AE, and will conclude the network is at fault rather than the disk.

And a volume reporting zero bytes free can still accept small appends for a while, because adding a couple of hundred bytes to a file whose last block has room needs no new allocation. The first write failure is later than the moment the disk filled.

10.20.5 What was not tested

A destination directory losing write permission. It could not be injected with `chmod`: the engine holds the output file open, and changing a file's mode does not affect a descriptor already opened against it. Testing it properly needs the volume remounted read-only or removed underneath the process, which has not been done. It is named here rather than omitted, because a gap nobody mentions reads as a gap nobody looked for.

11 Alerting

Nothing outside Perfuse will tell you a feed has stopped. The sender is being acknowledged and is satisfied; the receiver has no way to know a message was sent. Alerting is the only thing standing between a broken interface and somebody noticing weeks later.

11.1 The alert kinds

Kind	Fires when
error-rate	Failures exceed a proportion of traffic.
queue-depth	A queue has more than N messages waiting.
queue-age	The oldest queued message is older than a duration.
queue-stuck	A queue is not draining at all.
no-traffic	Nothing has arrived for a fixed period.
below-rhythm	Traffic is well below what this feed normally does *at this time of week*.
channel-down	A channel that should be running is not.
script-errors	A script is failing.
rows-quarantined	A database source is setting rows aside.
slow-delivery	Deliveries are succeeding but taking too long.
contract	A message violates the channel's declared contract.

11.2 Silence is the hardest thing to detect

A channel that has stopped receiving produces no errors. There is nothing to count. This is why no-traffic exists, and why it is not good enough on its own.

no-traffic is a fixed threshold: alert if nothing has arrived for, say, two hours. On a feed that genuinely never goes quiet, that works. On a clinic's laboratory feed it is useless — the feed is silent every night and all weekend, so a two-hour threshold pages somebody at 1am every night, and a threshold long enough to survive the weekend will not notice a Tuesday morning outage until Tuesday afternoon.

There is no single number that is both. That is not a tuning problem; it is that the question "has this feed stopped" cannot be answered by a constant when the feed's normal varies by a factor of a hundred

across the week.

11.3 Rhythm-aware detection

below-rhythm compares current traffic against what this channel normally does **in this hour of this day of the week**.

Perfuse learns each channel's week from its own recorded history: 168 hourly buckets, Monday first so a working week is contiguous. For each bucket it holds a median, a low and a high, a floor, and how many observations it is based on.

The threshold is a share of normal, defaulting to 0.8 (80 per cent in the console) — alert when traffic falls below four fifths of what is normal for this hour.

11.3.1 Median, not mean

A single backlog flush of forty thousand messages against a normal three hundred moves a mean far enough that the feed can go completely silent afterwards without ever falling below it. The median is unmoved by one extraordinary hour, which is exactly the property needed.

11.3.2 The floor is a percentile, not the minimum

Whether a channel's rhythm is reliable enough to alert on is decided by its low end, not its high end. A shortfall alert fires when traffic is **low**, so only the low end can tell you whether it will produce false positives. An early version of this compared the high against the median and got it wrong in both directions at once: it disqualified a perfectly steady feed for a month after one backlog, and it passed a genuinely erratic feed that swung between two and nine hundred messages in the same hour because its high was only three times its median.

The floor is the twentieth percentile rather than the minimum, because the minimum is dragged to zero by a single bank holiday. And the percentile is not interpolated — every number in an alert at two in the morning should be a count the feed actually produced, not an average of two counts it did not.

11.3.3 It will say when it cannot tell

A channel with no history at all is omitted entirely rather than treated as one that has stopped. A new channel must not look like a broken one.

Where the history is too thin or too erratic to support a judgement, the rule declines to fire rather than guessing. "I cannot honestly say" is a first-class answer here, and it is preferred to an alert that is right slightly more often than chance.

Absent hours inside the learning window are recorded as real zeroes, not as missing data. Half the signal in a clinic feed is **when it is legitimately quiet**, and treating those hours as unknown would throw it away.

11.4 Setting alerts up

From the interface, under Alerts, per channel or across all of them. Each rule has a kind, a threshold and where to send the alert. The rules are saved to `alerts.yaml` beside the database (or the file `-alerts` names) and take effect without a restart; until the first save the built-in rules apply.

Worth doing before a feed goes live rather than after the first incident. The specific set worth having on any clinical feed:

1. `below-rhythm` — the feed has gone quiet when it should not be quiet.
2. `error-rate` — messages are arriving and failing.
3. `queue-age` — something is stuck rather than merely busy.
4. `channel-down` — the channel is not running at all.

The first is the one that catches the failure nobody else will report.

11.5 Where alerts go

Alerts can be delivered by email, by HTTP to whatever you already use for on-call, or written to the log for a collector to pick up.

An alert that is only written to a log nobody reads is not alerting. If Perfuse is the only thing that will notice a stopped feed, the alert has to reach a person.

12 The Message Store

Every message Perfuse receives is recorded, as received, before anything modifies it. So is what was sent to each destination and what happened to it. The store is a SQLite file given by `-db`.

12.1 What is kept

Per message: the raw bytes as they arrived, when they arrived, which channel, the parsed message type and trigger event, and the filter outcome.

Per destination per message: the bytes that were sent, the outcome, the time taken, and any error or negative acknowledgement received.

This is the shape an investigation needs. "What did they send us", "what did we send them" and "what did they say about it" are three different questions and each of them gets asked.

12.2 Searching

By channel, by time range, by outcome, by message type and trigger event, and by field value.

Field search works on the parsed message, so searching for a record number finds it wherever in the message that field lives, rather than matching the digits anywhere in the text. A text search for 12345 in a message containing a quantity of 12345 finds a message that has nothing to do with the patient you are looking for.

12.3 Replay and reprocessing

Because the original is kept, a message can be run through the current configuration again.

Trace takes a recorded message and shows what the channel would do with it now, without sending anything. Covered in debugging.

Reprocess actually re-runs it, including delivery. This is what fixes a batch of messages that failed because a destination was misconfigured: correct the configuration, select the affected messages, reprocess.

Reprocessing delivers again. If the messages partly succeeded the first time, reprocessing will produce duplicates at the destinations that already had them. Perfuse records which destinations succeeded, so reprocessing can be limited to the ones that failed — do that rather than reprocessing everything, unless the receiving system deduplicates on message control ID.

12.4 Staleness

A trace of an old message against the current configuration is answering a hypothetical: what *would* happen now. That is usually what you want, but not always — if you are investigating what went wrong last Tuesday, the channel may have changed since.

Perfuse detects this and says so. A trace of a message that arrived before the channel was last edited is marked as such, so the answer is not mistaken for a reconstruction of what actually happened.

12.5 Retention

The store grows. `cleanup.periodDays` controls how long messages are kept.

Two things to weigh. Clinical messages may fall under retention requirements that are longer than anything you would choose for operational reasons, and the store is not the system of record — the receiving system is. Perfuse's copy exists to diagnose and to reprocess, and both of those needs fall off sharply after a few weeks.

Retention also decides how much history the rhythm learner has to work with. See alerting: a retention window shorter than a few weeks leaves it unable to distinguish a quiet Sunday from a stopped feed.

12.6 The store holds clinical data

This is the part of Perfuse that contains patient information, and it should be treated as such: on encrypted storage, backed up as clinical data, and access-controlled. See security.

The generated test messages described in testing exist partly so that this data does not have to leave the environment in order to reproduce a problem elsewhere.

13 Debugging

The question is almost always the same: this message came in, something wrong went out, where did it change? Perfuse answers it by replaying a recorded message through the current configuration and showing every step.

Nothing is sent while tracing. The channel keeps running untouched, and no redeploy is needed.

13.1 Tracing a message

Select any recorded message and trace it. The result shows, in order:

- **What the filter read.** Which fields it looked at and what values it found. A filter rejecting everything is usually reading a path that does not exist in this feed, and this names it.
- **Each transformation.** What it changed, from what to what.
- **What each destination decided.** Whether it would accept the message, and why not if it would not.
- **The output.** The message as each destination would send it.

13.2 Step by step

The step-through view runs each transformation on its own and snapshots the whole message after it. A numbered rail of steps, coloured by what happened, and the message as it stood at each point.

The whole message rather than only the change, because the change is often not where the problem is — a step that correctly modified the field it was aimed at, on a message where an earlier step had already put the wrong thing there, looks perfectly correct in isolation.

13.3 Four outcomes, and the distinction that matters

Outcome	Meaning
changed	The step ran and modified the message.
no-effect	The step ran, and the message is unchanged.
skipped	The step's when condition was false, so it did not run.
failed	The step errored.

no-effect and skipped are reported separately, and this is the point of the whole feature.

From outside they are identical: the message is the same afterwards either way. They mean opposite things. `skipped` means the condition worked as written and this message was not one it applied to —

usually correct. no-effect means the step ran and found nothing to do, which usually means it is aimed at a field that is not there.

An engine that reports both as "nothing happened" leaves you unable to tell a working conditional from a broken path, and that is the most common transformation bug there is.

A step reporting no-effect on every message is nearly always a wrong path. The step-through header counts how many steps found nothing, which is the fastest way to spot it: a channel with three steps and three no-effect results is not transforming anything at all.

13.4 Testing a change before saving it

The builder can run a candidate configuration against recorded traffic and report what would differ. This is the safest way to change a live channel: make the edit, see what it does to the last few hundred real messages, then save.

It is also how to find out that a change which looks obviously correct affects messages you had not thought about — a step conditioned on one trigger event, on a feed that turns out to carry four.

13.5 What tracing cannot tell you

It runs the current configuration. If the channel has been edited since the message arrived, the trace is a hypothetical rather than a reconstruction, and Perfuse marks it as such rather than letting the distinction pass silently.

It also does not exercise the transport. A trace shows what would be sent, not whether the destination would accept it — a message that is correct and a destination that is refusing connections produce a clean trace and a failed delivery. Those are visible in the delivery record instead.

13.6 Scripts

A script is one opaque step in the trace. Perfuse can show what went in and what came out, and cannot show what happened in between or which fields were read.

This is the concrete cost of using a script rather than declarative steps, and it is worth weighing when choosing. See transformation.

14 Testing

Standard advice for testing an interface is to build sample messages that reflect your own environment — your laboratory's codes, your case mix, your sender's habits — and to strip the patient data out by hand. That is slow, and doing it incompletely is the normal outcome.

Perfuse offers three things instead: generated traffic that matches a feed's shape without its content, a channel proposed from a sample somebody sent you, and a comparison against the engine you are replacing.

14.1 Generated test traffic

If a channel has handled messages, Perfuse can produce more messages like them.

What is faithful:

- Trigger events in the proportions actually observed, with at least one of each so a rare event still appears.
- Fields populated at the rates they really are — so the case where a field is **absent** gets tested, which is the case that breaks things.
- Repetitions where repetitions occur, composite structure where it occurs.
- Codes drawn from the code tables actually seen.

What is not faithful, and this is stated rather than implied: **clinical coherence**. A generated patient has an unrelated record number, a name-shaped string, and a diagnosis with no relationship to the observation. These messages exercise an interface. They are not a clinical test set and cannot be used to validate clinical logic.

14.1.1 Why it contains no real data

The argument is structural rather than a claim about effort. The generator's only input is the channel's traffic **profile**, and a profile holds fill rates, shapes, lengths, repetition counts and segment frequencies. It holds no values.

The single exception is code-table fields, and a code identifies nobody.

This is a stronger guarantee than de-identifying real messages, which is a process that can be done incompletely and usually is. Here there is no path by which a name could reach the output, because the name was never in the input.

Every generated message is parsed before it is returned — an unparseable corpus would send somebody hunting for a bug in their interface that is really a bug in the generator — and every one is marked T in MSH-11, with a deliberately synthetic sending application and facility.

A test message that is not marked as one will eventually be found in a production system by somebody who has no way of knowing it was not real. The T and the synthetic sender are not cosmetic.

14.2 Building a channel from a sample

Paste the message a laboratory, hospital or vendor emailed you and Perfuse proposes a channel for it. Nothing needs to be running first.

It handles what people actually paste: several messages in one block, any line ending, MLLP framing that survived the copy, and the covering note above the message. Refusing a sample because it arrived with Windows line endings would be an obstruction rather than a check.

14.2.1 Three lists

The proposal comes with what the sample **shows**, what has been **guessed**, and what the sample **cannot tell you** — three separate lists, weighted equally in the interface.

They are separate because a reader who cannot tell which is which has to verify everything or trust everything, and both are worse than knowing where to look.

From a single message the limits are named specifically rather than generally: which fields are optional, which segments repeat, which codes exist beyond the ones here, whether the sender ever sends a different trigger event. A general caution is one people skim.

Confidence is a word — low, moderate, reasonable — never a percentage. A percentage implies a calculation, and this is a judgement about how many samples there are.

Z-segments are called out by name, because a local segment is the thing no specification mentions and the commonest reason a channel built from a standard template does not work.

14.2.2 The separator warning

If the sample's field separator is not the usual |, that appears **first**, and confidence drops to low.

The reason is a genuine silent failure. HL7 takes its field separator from the message itself, so a sample whose pipes have been replaced by typographic look-alikes — which is what a word processor does — parses perfectly into fields that are all wrong. Nothing errors anywhere.

Refusing such a sample would be wrong, because a sender may legitimately use another separator. So it is flagged, the character is named, and asking for a plain text attachment is suggested.

14.3 Comparing against the old engine

Before moving a live feed, the question is whether Perfuse produces the same output as the thing that has been running for years. No amount of design quality answers that. See migration.

14.4 Verifying against the real thing, in containers

Three checks run against real implementations rather than against Perfuse's own idea of a protocol. They live outside `make` check because they need a container runtime, and a check that needs Docker is a check people stop running.

`scripts/interop-up.sh` starts all of them and prints what each one unlocks.

Container	Checks	What it found
Keycloak 26	web/e2e-saml/, and internal/saml fixtures	A defect. The canonical form a signature covers was computed with namespace prefixes dropped, so no real identity provider could ever have signed anybody in
Microsoft Entra	internal/saml live tests and fixtures	Two defects, both in what surrounds the protocol. An account was created under Entra's opaque NameID, because the readable-name logic wanted an email claim and Entra sends none. And a SAML account was recorded as OIDC, because the store derived the protocol from an issuer that looks identical for both
HAPI FHIR	go test ./internal/engine/ - run HAPI	A defect. An unmapped assigning authority produced an identifier system announcing an OID and carrying a word, which HAPI accepted without complaint
ActiveMQ	go test ./internal/engine/ - run RealBroker	Correct, including the null-byte framing case. Asserted through the broker's own management interface rather than through a function returning nil
nginx, client certificates	go test ./internal/engine/ - run ClientCertificate	Correct
Orthanc (a real PACS)	go test ./internal/engine/ - run RealPACS	Correct. A C-STORE arrives and the patient name, identifier, study date, modality and SOP class all survive
PostgreSQL	go test ./internal/engine/ - run RealPostgres	Correct, and it prompted a better error: a statement in the wrong dialect's placeholders is now refused when saved rather than becoming a syntax error at the server when a message arrives
OpenSSH sftp-server	go test ./internal/engine/ - run OpenSSHServer	Correct. The existing SFTP tests use the same Go library on both ends; OpenSSH is a different implementation and the one on the far end of nearly every real feed
Mirth 4.5.2	go test ./internal/mirth/ -run RealMirth	Correct now. The only fixture used to be hand-written and Mirth would not load it; scripts/mirth-author-channel.sh has Mirth author one instead, and Perfuse reads it

Every one skips rather than fails when its container is absent, so make check passes on a machine with no Docker.

Why this is separate from the rest of the suite. Every other test of these features had both halves written here: a document this codebase signed, read back by the code that signed it. That is agreement with oneself, and it is worth less than it appears.

The SAML canonicaliser is the case that proves the point. It carried a thousand lines of tests, five of them specifically about canonicalisation, and all of them passed while **no real identity provider could ever have signed anybody in** — the canonical form that a signature covers was being computed with namespace prefixes dropped, and the only way to find that was to hand it a document Perfuse had not written.

Two of the eight checks found defects outright, and a third prompted a real improvement. Three found nothing wrong. That ratio is the argument for keeping all of them: a check that passes has told you something you could not otherwise have known, and there was no way to tell in advance which ones they would be.

One pattern worth naming, because it appeared in both defects: **acceptance is not verification**. HAPI accepted the malformed identifier system and stored it; the defect only appeared on reading the resource back. A receiver that tolerates something is not evidence that it is right, only that this receiver was lenient.

Two captured assertions are committed, at `internal/saml/testdata/keycloak-assertion.xml` and `entra-response.xml`, with tests over both — so that particular regression is caught without a container.

Both are needed rather than one being spare. Keycloak prefixes its assertion and signature, Entra does not, and those are the two opposite cases in canonicalisation — which is precisely where the defect was. A test asserts each document still covers its own case, because a replacement capture that happened to be prefixed would quietly halve the coverage without failing anything. The general lesson does not generalise so cheaply: for anything else, the container is the check.

14.5 Diagnosing a test that fails once and never again

A test that fails in a full run and passes alone sixteen times is not a flaky test. It is a test that shared something with the run, and the way to find out what is to measure the shared thing rather than the test.

Two instruments exist for this, and both are meant to be run alongside a full suite:

```
./scripts/e2e-watchdog.sh /tmp/watchdog.log      # the HTTP server and the shared
session
./scripts/e2e-mlp-probe.py /tmp/mlp.log          # the fixture channel's listener
```

Each discovers the run's ports from the setup's own state file and samples every 500 milliseconds until the server exits. They log every sample rather than only the failures, which is what makes them useful: a blip shows up in a run where nothing failed, so a fault that appears once in three runs becomes measurable on every one.

Set `PERFUSE_E2E_KEEP=1` to keep the temporary directory, which holds the database as it was left and the server log for the whole run. **The server log is usually the answer.** In one investigated case it

recorded a channel stopping and its listener rebinding 421 milliseconds later, and the probe's one refused connection fell inside that window.

Read the trace rather than `error-context.md`, which has been empty on more than one occasion where the trace held the real message.

Copy the artefacts out before re-running anything. A passing run clears `test-results`, so the trace of the failure you are investigating disappears the moment you try to reproduce it. Once lost, the question it would have answered cannot be answered.

Stop the interop containers first. Stop them before a full browser run, and start them with `./scripts/interop-up.sh` afterwards if you still need them.

Eight containers hold around 9.8GB of memory, and on a machine without much more than that there is too little left for Chromium's processes, node and the Go toolchain together. What it produces is not an out-of-memory error but a renderer that stops being scheduled: one recorded trace shows a **60.2 second gap with no frames at all**, during which Playwright waited for a button to become "visible, enabled and stable" and could not get two consecutive animation frames to compare. The server was healthy throughout, the slowest request in the whole run being 154 milliseconds.

That failure reads as a defect in whatever test happened to be running. It is worth knowing the shape of it: every request fast, the page visually frozen, the retry loop never reaching a second iteration, and the whole thing passing in isolation.

Only one run at a time. The suite shares one server, one signed-in session in `web/e2e/.auth/admin.json`, and one state file naming the server's process. None of that is per-run, so a second run starting while the first is going overwrites the state file, and whichever teardown finishes first kills the other run's server and deletes its session.

What that looks like is nothing like the cause. It produced a run of 279 failures whose first message was `ENOENT: no such file or directory, open './e2e/.auth/admin.json'`, a run that stopped early reporting 217 passed, and two tests failing with `bind: address already in use` — none of which suggest two runs fighting. The suite now takes a lock and refuses to start, naming the process that holds it. A lock left by an interrupted run is taken over rather than blocking for ever, because a safety measure that wedges the suite is one somebody deletes along with the safety.

A related trap, now fixed: two specs named their listening ports outright, so a server left behind by an interrupted run made them fail on something they had nothing to do with. Ports are asked of the operating system now.

Do not answer any of this with retries. A test that passes on the second attempt found a real race and hid it, and `playwright.config.ts` sets `retries: 0` for that reason.

One caution: sampling every 100 milliseconds spawned twenty processes a second, stretched a run by half again, and failed two tests on timeouts that had nothing wrong with them. An instrument that changes what it measures produces findings about itself.

14.6 What still needs real infrastructure

Several parts of Perfuse are implemented and have never been exercised against the real thing. They are listed here rather than left to be discovered:

- SMB against a real Windows share — the protocol code is complete and tested for path safety and argument handling, but no test opens a socket to a domain-joined server.
- A real serial port.
- The Mirth importer against XML a real Mirth produced. Tested against Mirth 4.5.2 in a container, and the finding was about this repository: the only migration fixture here was written by hand and **Mirth will not load it** — it stores the channel as invalid and discards every connector. What was settled is that the version attributes real Mirth writes on every element parse to the same channel, so they are not a hazard; what is not settled is everything else a real export contains.
- SAML against a hosted commercial identity provider. Keycloak is verified; one provider proves an implementation matches one vendor.
- An NCPDP claim through a real pharmacy switch.
- A SCRIPT message through Surescripts, where certification is a commercial process rather than a technical one.

The features in this chapter and the next exist so that you can gather your own evidence rather than taking anybody's word.

14.7 Measured against public conformance corpora

A test suite whose fixtures were written here proves agreement with oneself. These are other people's data.

14.7.1 FHIR R4, the specification's own examples

All 2,912 example files published with the FHIR R4 specification — every example the authors of the standard wrote.

Result	Count
Resources validated with no findings	13,723
Resources reported invalid	0
Files refused as an unimplemented resource type	753
Bundle entries skipped as an unimplemented type	1,348

Not one resource was validated incorrectly. Every failure is an explicit refusal naming the type it cannot read — 672 ValueSet, 80 ConceptMap and one Parameters. A validator that quietly passes what it does not understand is worse than one that says so.

The corpus found a defect, which is the reason for running it. Validating a bundle printed "Bundle validation from a file is not supported yet" and carried on. That was false in both directions: bundles whose entries happened to be types the parser could hold were already being validated, and a bundle that was genuinely invalid produced the same line, counted nothing, and exited zero — including under `-strict`, which exists so this can gate a build. Forty-two of the specification's own bundles were being skipped.

Bundles are now validated entry by entry, with three outcomes kept apart: an entry that is wrong fails and names the field, an entry of a type this build cannot read is counted as skipped and reported, and a bundle carrying no resource bodies is neither. The same corpus went from 2,100 resources checked to 13,723.

14.7.2 HL7 v2, the HAPI test corpus

The message fixtures from HAPI, the reference Java HL7 v2 implementation — deliberately awkward material including uuencoded payloads, escaped delimiters and repeating groups.

59 messages, 59 parsed, nothing refused. Seven message types in a corpus any single system would describe as one thing, which is the point `perfuse profile` exists to make.

14.7.3 What has not been run

DICOM against a public conformance set, and X12 against a published corpus. Neither has been done, and neither should be inferred from the two above.

15 Migration from Mirth

Two things are needed to replace a working engine: the channels have to come across, and somebody has to be able to show that the replacement produces the same output. Perfuse does the first by importing Mirth's own exports, and the second by comparing the two engines on your own traffic.

Everything in this chapter applies equally to the Open Integration Engine and BridgeLink. Mirth Connect stopped being open source at 4.6 in March 2025; OIE (hosted by the Eclipse Foundation) and BridgeLink (Innovar Healthcare) are the forks that continue the open-source line, and both write the same export formats. The importer is tested against all three, below.

15.1 Importing channels

Perfuse reads every export these engines write:

Export	From the Administrator	What comes across
A channel	Export Channel	The channel
A channel group	Export Group	Every channel in it, each keeping the group
Code templates	Export Code Templates	Each library as one script file
A server backup	Backup Config, or GET <code>/api/server/configuration</code>	All of the above at once

Point `perfuse translate` at any of them, or drop one on **Administer → Migrate**:

```
perfuse translate -o ./channels server-backup.xml
```

A **code template library** becomes `lib/<library>.js` beside the channels, and every channel Mirth had the library enabled for gets `scripts.include: [lib/<library>.js]` — so a transformer calling a site function such as `formatMRN()` still finds it. A channel the library was not enabled for does not include it, because its scripts could not see it in Mirth either. Drag-and-drop snippets are left out: Mirth only ever pastes those into a step by hand, so they are not code that runs on its own. In the browser, importing a channel saves the libraries it includes first.

A **Channel Writer** names the channel it delivers to by id. In a group or a backup the target is in the same document, so it becomes a `channel` destination naming that channel; from a single-channel export it is reported for you to fill in. Server-wide global scripts have no Perfuse equivalent and are reported rather than dropped.

The report says, per channel, whether it translated cleanly, translated with warnings, or could not be read.

The headline is a fraction with its denominator — "37 of 40 ready" — because "37 ready" tells you nothing about what you still have to do.

A channel that could not be **read** is a different problem from one that could not be **translated**. The first means the XML itself did not parse and is usually a truncated or partial export; the second means the channel's content has something without a Perfuse equivalent. The import separates them.

15.2 What translates and what needs a decision

Sources, destinations, filters and the ordinary transformer steps translate directly. The connectors that translate: TCP/MLLP (and raw TCP with its framing), HTTP, File, Database, JavaScript, DICOM, Web Service (SOAP), SMTP and Channel Reader/Writer. JMS and the Document Writer do not, and are reported as blockers.

A Mapper step that sets a variable puts it into the map its scope names — `channelMap` by default — exactly as Mirth does, so later steps reading it with `$( 'name' )` find it.

JavaScript transformers do not, in general. Mirth channels commonly hold substantial code in a transformer, and there is no automatic conversion from arbitrary JavaScript to declarative steps. The importer reports these rather than attempting a translation that might be subtly wrong: a mistranslated transformer is worse than one flagged for a human, because it will run.

Where a script genuinely has to remain a script, Perfuse can run it — see the `scripts` key in the channel reference — at the cost of the step-level trace described in debugging.

15.3 What the importer has been tested against

Four engines, each authoring its own documents: Mirth 4.5.2, OIE 4.5.2 and 4.6.0, and BridgeLink 26.9.0. `scripts/mirth-engine-corpus.sh` compiles a small program against each engine's own jars and has its own serialiser write three ordinary channels — an MLLP feed with a mapper, a script calling a code template library and a filter; an HTTP feed into a database; a file poll out over MLLP — plus the library and a channel group. It loads them into the running engine and downloads the engine's own server backup. The results are committed under `internal/mirth/testdata/engines/`, and the test suite holds the importer to every one of them on every build: each backup is read, translated, loaded, and the ADT channel is run with two messages to check the library pads the record number and the filter drops the result.

Perfuse's own exports go the other way — Perfuse channels imported into each running engine and checked that the engine did not store them as invalid. All four accept every transport Perfuse exports.

The first run against this corpus found four defects, all fixed: the translator refused HTTP and File sources as "only MLLP is supported" long after Perfuse had both; it read an HTTP Sender's address from a field none of the engines writes; it translated Mirth's socket buffer size as a 64 KB message limit; and a Mapper's variable became a JavaScript global that `$( 'mrn' )` never saw, so the library padded an empty string.

The single-channel fixture that started this was also exported by Mirth 4.5.2, authored by Mirth itself rather than written here. That distinction earned its place: the only fixture in this repository used to be hand-written, and a real Mirth **will not load it** — the API accepts the POST and then stores the channel as `This channel is invalid`. Verify all required extensions are loaded correctly, with every destination connector discarded.

Hand-writing one was never going to work, and the reason is worth knowing if you are producing exports of your own. Mirth's serialiser ignores elements it does not recognise, so a wrong element name produces no error whatsoever — the connector simply is not there afterwards. There is nothing to read and nothing to correct against.

`scripts/mirth-author-channel.sh` builds one using Mirth's own model classes and its own serialiser, then posts it to a running server and fails if the server rejects it. Perfuse's importer reads the result: name, source transport and destinations all survive.

15.4 Proving it matches

This is the part that decides whether a migration happens, and it needs evidence rather than confidence.

Perfuse compares its own output against the old engine's, message by message, on your traffic. What it needs is **pairs**: the message as it arrived, and the message the old engine produced from it. Both are in Mirth's own message store and can be exported from its message browser.

Perfuse cannot reach into Mirth, and this manual will not pretend otherwise. A tool that claimed to verify a migration without ever seeing the old engine's actual output would be verifying nothing. Supplying the pairs is the honest shape of the exercise.

15.4.1 Four numbers, not a pass rate

Result	Meaning
Identical	Byte for byte the same.
Cosmetic only	Differs in ways that carry no information — trailing empty fields, a final terminator.
Differ	A field's value is different.
Would not run	Perfuse could not process the message at all.

They are four numbers because they mean four different things and collapsing them hides the ones that need a decision.

Identical means byte for byte, not equivalent. A weaker test would let differences in segment order or trailing separators through, and those are exactly what a fussy downstream system rejects.

Cosmetic differences are counted separately rather than folded into the good column. They genuinely carry no information, so treating them as failures would bury real findings under thousands of nothing — but they are not identical either, and a receiver doing its own strict parsing may disagree.

"Would not run" is separate from "differ" because it is usually a configuration gap the importer could not fill, not a mapping decision. If everything is in that column, the verdict says so and tells you to look at the configuration before reading anything else.

15.4.2 Differences are grouped by field

A run over fifty thousand messages reporting four thousand differences has told you nothing you can act on. The same run reporting that PID-8 differs in every message because the old engine wrote MaLe where Perfuse writes M has told you exactly one thing to decide.

So each finding is a field, with a count, and it says whether the difference is **systematic** — the same value pair every time — or whether the field's content differs.

That distinction is the most useful thing in the report. One distinct value pair across four thousand messages is a single mapping decision. Four thousand distinct pairs is the content differing, which is a different and much worse problem, and the two must not read alike.

Findings are ordered by how many messages they affect, so the largest single cause is first.

15.4.3 The verdict does not round

It gives exact counts: 49,993 of 50,000, not "over 99%". The seven are the whole point of running it.

It is written to be quotable in a change request, because that is where the number ends up.

15.5 A suggested sequence

1. Import the channels and resolve anything reported as unreadable.
2. For each channel, review the warnings. Most will be scripts.
3. Export a few thousand message pairs from Mirth for the busiest channel and run the comparison.
4. Work through the findings. Systematic ones are one decision each.
5. Re-run until the only differences are ones you have decided are correct.
6. Run Perfuse in shadow alongside Mirth on live traffic, delivering nowhere, and compare again over a period that includes a weekend and a month end.
7. Cut over one channel, not all of them.

15.6 What only runs on one vendor's software

The migration screen answers a second question, and it is worth asking before you have any opinion about Perfuse: how much of your integration logic can only run on the software you have.

Every channel gets a **Portability** verdict, from the same scan that produces the migration notes. It runs against a channel export, so it needs nothing installed and changes nothing.

15.6.1 The distinction is the whole point

Most Java in Mirth scripts is not lock-in. `SimpleDateFormat`, `HashMap`, `Apache Commons` — all of it working around a JavaScript engine from 2009, all of it ordinary, all of it with direct equivalents. A channel with three hundred of those references is fully portable, and a tool that reported "300 Java references, you are trapped" would be lying to you.

Lock-in is specifically the calls that need the vendor's own server: `com.mirth.connect.*` and vendor jars. Five of those and you are stuck. Five hundred of the other kind and you are free.

So the verdict leads with the number that matters and says in the same sentence that the larger number is not it. Where there are vendor calls, they are listed by name — a count is an assertion, and a list is something your own engineer can go and check.

A third category is kept separate: capability that exists but not from a script. Starting a channel from JavaScript has no equivalent here, and there is an API endpoint that does exactly the same thing. Calling that lock-in would overstate the finding.

15.6.2 External scripts are reported as unknown

A step that runs an external script file references a path on the old server, and the file is not in the export. Nothing can be said about what it contains, so it is counted as unknown rather than assumed either way. Fetch those files before concluding anything — in both directions.

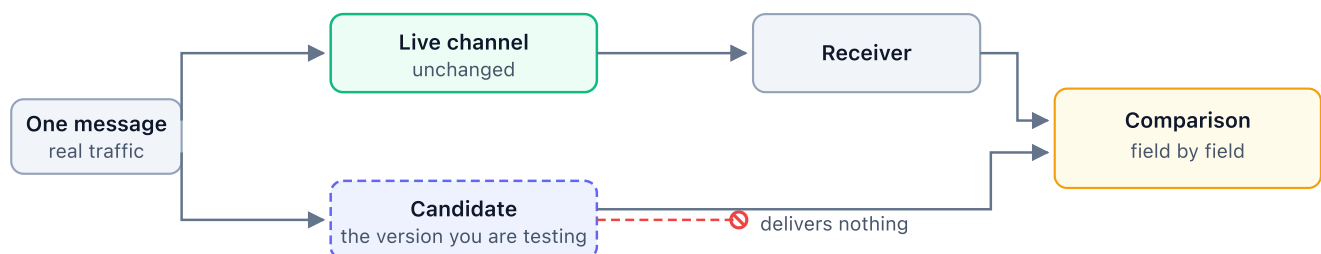
15.6.3 Perfuse's own lock-in, stated in the same breath

This section would be dishonest without it.

Perfuse is Apache 2.0, which is irrevocable for code already released. You can fork what exists today and nobody can withdraw that. Channels are YAML files and scripts are ordinary JavaScript, so there is no proprietary surface to call — Mirth's flavour of Java dependency is structurally unavailable here, which is the one place a limitation is genuinely a feature.

What that does not promise: that a future version stays open, that a hosted service exists at a price you like, or that migrating away from Perfuse costs nothing. Moving any integration layer means re-testing every channel against real traffic. What you are choosing is whether the artefacts you build are readable and portable, and whether the exit depends on a licence somebody else controls.

15.6.4 Setting up step 6



The candidate sees the same real messages as the live channel and delivers none of them. Only the live channel reaches the receiver. What you read afterwards is where the two disagreed, field by field — not a pass or a fail.

The **Shadow** section starts a comparison. Choose the channel that is running, choose the candidate to compare against from the list, and start.

Three things about it are worth knowing before you rely on the numbers.

The candidate is chosen from a list, not typed. A shadow names a channel file, and a name that does not resolve makes the **live** channel invalid — Perfuse refuses a channel wholesale when anything it references is broken. So a typo does not produce a warning, it makes the running channel disappear.

The share is a percentage. Lower it only on a busy feed, and know what it costs: sampling reduces work and confidence in the same proportion, and the message that would have shown the difference is the unusual one.

Fields to ignore is usually needed. A channel that stamps a timestamp or a sequence number differs on every single message, which reports a difference rate of 100% and tells you nothing. MSH-7 and MSH-10 are the common pair.

A comparison is written to the channel file as soon as you start it, and begins observing when the channel next loads — the screen says so rather than claiming it is already running. Stopping a comparison leaves the candidate channel in place, because the candidate is the thing meant to go live.

Step 6 is the one that cannot be shortened. The differences that matter are usually in traffic that only appears occasionally — a rare trigger event, a month-end batch, a sender that behaves differently at midnight — and a sample taken over an afternoon will not contain them.

15.7 Going back the other way

A channel can be exported as a Mirth channel file, from the browser: **Channels**, then **To Mirth** on the channel you want.

This exists because the objection to replacing a working engine is rarely whether the replacement is any good. It is what happens if the decision turns out to be wrong, and an export that goes back the way it came is the cheapest answer to that.

Read what the dialogue tells you before you take the file. Mirth's format has nowhere to put a Perfuse filter, a transformation chain, a contract or a shadow comparison, so an exported channel does less than the one it came from. The transports and their addresses convert; the logic does not. Each loss is listed by name before the file is offered, and the same list goes into the channel's description so that whoever imports it sees it too.

A channel Mirth cannot express is refused rather than approximated. An S3 destination has no Mirth equivalent, and a channel that imported cleanly while delivering somewhere other than the bucket would be worse than no file at all. The refusal names the destination at fault.

The connector definitions are not written from a reading of Mirth's format. `scripts/mirth-dump-connector-templates.sh` has Mirth's own serialiser produce the defaults for each connector class, and Perfuse substitutes values into those documents — refusing any setting the template does not already contain. This matters because of how Mirth answers a channel it cannot parse: it does not refuse the import. It stores a channel whose description has been replaced with "This channel is invalid. Verify all

required extensions are loaded correctly" and whose destinations have been silently discarded. An exporter checked against its own output rather than against a server passes every test while producing documents that are thrown away, which is exactly what happened here before the exports were tried against a running Mirth.

What crosses today: MLLP and TCP in both directions, HTTP in both directions, file reading and writing, database reading and writing, DICOM in both directions, and SMTP, SOAP and JavaScript destinations. Each pair is verified by importing into a real Mirth 4.5.2 and checking the server did not quietly replace the channel.

16 CMS-0057 for payers

The CMS Interoperability and Prior Authorization final rule (CMS-0057-F) requires impacted payers — Medicare Advantage organisations, state Medicaid and CHIP programs and their managed care plans, and QHP issuers on the federal exchanges — to run four FHIR APIs from 1 January 2027, and to post prior authorisation metrics on their public website every year from 2026.

Perfuse provides the four APIs on its FHIR endpoint and the conversions that fill them: adjudicated claims as CARIN Blue Button, prior authorisations as Da Vinci PDex, and the yearly metrics in the layout of CMS's own template. The **CMS-0057** view, under Exchange, says which of the four this instance can answer and what each still needs, and does all three conversions in the browser.

Drugs are out of scope throughout, as they are in the rule's prior authorisation provisions: no pharmacy claims, no formulary, and a metrics row naming a drug is refused rather than counted.

16.1 The four APIs

API	What it serves	Guides	Turned on by
Patient Access	A member's claims, clinical data and prior authorisations, to an app they choose	CARIN Blue Button 2.2.0, PDex 2.2.0, US Core, SMART App Launch	-fhir and a SMART issuer (-smart-issuer)
Provider Access	The same for a provider's attributed members, in bulk	PDex 2.2.0, Da Vinci ATR 2.1.0, Bulk Data	-fhir-payer-apis and -fhir-bulk-export
Payer-to-Payer	A member's record to another payer, after matching them	HRex 1.2.0 \$member-match, PDex 2.2.0	-fhir-payer-apis
Prior Authorization	What an order needs, the documentation, then the request and decision	Da Vinci CRD 2.2.1, DTR 2.2.0, PAS 2.2.1 with X12 278 translation	-fhir, writable, and -crd-rules

The Patient Access API needs no operation of its own: it is the FHIR endpoint's ordinary read and search, protected by SMART, with CARIN and PDex resources in the store. Searching ExplanationOfBenefit supports the parameters CARIN names — patient, identifier, type, service-date, billable-period-start, provider, insurer, coverage — and Coverage supports subscriber-id, identifier and payor.

16.2 Coverage requirements and documentation: CRD and DTR

The Prior Authorization API is in practice three Da Vinci guides together. **CRD** tells the clinician, while the order is still a draft, whether it is covered, whether it needs prior authorisation and what

documentation is wanted. **DTR** hands the EHR the questionnaire that gathers that documentation. **PAS** submits the request and returns the decision. Perfuse serves all three as the payer.

`-crd-rules rules.yaml` serves CRD over CDS Hooks at `/cds-services`: the `order-sign`, `order-select`, `order-dispatch` and `appointment-book` hooks, each with a prefetch for the patient's active Coverage. The rules are the payer's policy, kept in a file because they change without software releases:

```
payer: Springfield Health Plan
rules:
  - description: Home oxygen equipment
    codes: [E0424, E0431]
    covered: covered          # covered, not-covered, conditional
    pa: auth-needed          # no-auth, auth-needed, satisfied, performpa,
conditional
    documentation: [clinical]
    questionnaire: https://www.springfield-health-
plan.example/fhir/Questionnaire/home-oxygen
```

Each order gets a `systemActions` update carrying CRD's coverage-information extension - which the EHR stores on the order and later sends on with the prior authorisation - and a card the clinician reads. An order whose code has no rule is answered `conditional`, never assumed covered; a request with no Coverage at all gets a card saying so. `examples/crd/` has a rules file and a matching questionnaire.

A rule can also carry the rest of what CRD lets a payer say:

- `billing_codes`: the codes to bill under.
- `details`: limits and costs, such as `allowed-period` or `in-network-copay`.
- `contact`: who to call.
- `expiry_days`: how long the answer holds.
- `depends_on`: codes of other orders the answer depends on, so the EHR asks again if they change.
- `doc_purpose`: what the documentation is for (`withclaim`, `withorder`, `retain-doc`). `withpa` is refused, because CRD 2.2.1's

own invariant `crd-ci-q4` rejects it.

What the EHR did not prefetch, Perfuse reads from the EHR's `fhirServer` with the token it sent. If that fails, the answer is `indeterminate` with CRD's technical reason, never a guess. A Coverage that has ended or is not active is answered `not-covered`, `no-active-coverage`.

`members`: `fhir` in the rules file resolves the coverage against the payer's own records in this server's FHIR store (start it with `-fhir`), the same records `$member-match` reads:

- a patient who is not a member is answered `no-member-found`;
- a member whose coverage does not match is answered `coverage-not-found`;

- a prior authorisation already approved there for the service (a ClaimResponse with a preAuthRef) makes the answer satisfied,

with that number.

Discovery declares `davinci-crd.version` and a `coverage-info` configuration option. When a client turns the option off, the answer is empty. EHR JWTs are checked as CDS Hooks defines them, with `iss`, `aud`, `exp`, `iat` and `jti` and no `sub`, and a repeated `jti` is refused.

Calls authenticate with the JWT an EHR signs for each call, from an issuer named in `-cds-clients` with its JWKS URL - which is how CDS Hooks describes it - or with a Perfuse API token. Discovery is open, as CDS Hooks requires.

DTR: load the questionnaires into the FHIR endpoint, and `POST /fhir/Questionnaire/$questionnaire-package` returns them as DTR package bundles, with the CQL Libraries they name. The request can name the questionnaire, or send the order and let its `coverage-information` say which.

Trying it: Exchange → Claims & auth → Coverage requirements (CRD) sends an order to this server's rules, or to a payer's CDS service URL, and shows the cards and the coverage information.

16.3 Claims as CARIN Blue Button

`perfuse cms0057 carinbb` pairs each claim in an 837 with its payment in an 835, by the patient account number the 835 is required to echo back, and writes a FHIR transaction for each: the `ExplanationOfBenefit` with the Patient, Coverage, Organizations and Practitioners it references.

```
perfuse cms0057 carinbb -system https://fhir.your-plan.org/identifier -network
innetwork \
    claims.837 remittance.835
```

The profile follows the claim: professional claims (837P) become `C4BB-ExplanationOfBenefit-Professional-NonClinician`, and institutional ones (837I) inpatient or outpatient by the type of bill. Every entry is a PUT with an id taken from the source identifiers, so loading the same claim twice updates it rather than making a second copy.

Three things are not in either X12 file, and each is handled by saying so rather than by guessing:

- **The identifier system.** CARIN requires the member identifier to name the payer's namespace. – system is required; there is no default, because a default would give every payer using Perfuse the same namespace.
- **The network status.** Neither transaction says whether the billing provider is in network, and it decides what the member pays. Without `-network` the benefit payment status is other and the conversion notes why.
- **A claim the 835 does not mention.** A CARIN `ExplanationOfBenefit` is an adjudicated claim. Without the 835 there is no payer claim number, no payment and no member liability, so the claim is reported as skipped and not converted.

Diagnoses are written with their decimal point (J02.9, not J029), because that is how the code system publishes them and a terminology server asked about the other spelling says it does not exist. A dependent's member identifier is the subscriber's, which is what the 837 carries, and the notes say so. An NPI whose check digit is wrong is reported before US Core rejects it.

16.4 Prior authorisations as PDex

`perfuse cms0057 priorauth` converts a Da Vinci PAS ClaimResponse — or the PAS response bundle that carries one — into a PDex prior authorisation ExplanationOfBenefit, which is what all three data APIs serve.

```
perfuse cms0057 priorauth pas-response.json pas-claim.json
```

The PAS Claim is optional and nearly always needed. An ExplanationOfBenefit must say which coverage the decision was made under, and only the request carries that; without it the conversion stops and says so. The review action — approved, denied, pended, with its authorisation number and reason — moves from PAS's extension to PDex's, and the X12 service type keeps its code under the URI PDex binds it to.

16.5 The Provider Access and Payer-to-Payer APIs

With `-fhir-payer-apis` the FHIR endpoint answers three more operations:

- `POST /fhir/Patient/$member-match` — HReX member matching. The coverage identifiers narrow to a few candidates, and the family name and date of birth must then agree; a given name is compared by initial, and gender only when both sides state it. More than one surviving candidate is not a match, and the answer is 422 rather than a guess, because a guess discloses a stranger's claims. An active Consent permitting disclosure is required, since the Payer-to-Payer API is opt-in for the member; `-fhir-member-match-without-consent` turns that off for testing.
- `GET` or `POST /fhir/Group/{id}/$davinci-data-export` — the Da Vinci bulk export of a Group's current members. `exportType` chooses the audience: `hl7.fhir.us.davinci-pdex#provider-download` or `#provider-delta` for Provider Access, `#payertopayer` for Payer-to-Payer.
- `GET /fhir/Group/{id}/$export` — the standard Bulk Data group export, treated as Provider Access.

The exports apply the rule's exclusions on the way out. Claims lose the payment and every cost-sharing and remittance amount, and keep the services, codes, dates and providers; the CARIN profile is removed from `meta.profile` because the resource no longer meets it. Payer-to-Payer also leaves out denied prior authorisations. Members holding an active PDex Provider Access opt-out Consent are left out of a provider export, and the log records how many. The patient parameter can only narrow to members of the Group; naming anyone else is refused.

Access to a Group's export is decided by the endpoint's authentication. SMART scopes say what kinds of resource a token may read, not which Group, so each provider needs a token that can reach only its own attribution list. An API token can be limited to Groups when it is made — `perfuse token create -label riverside-provider-access -fhir-groups riverside-attributed`, or the **Limit to FHIR Groups** field under Users → Machine credentials — and on the FHIR endpoint it can then read those Groups, run

\$davinci-data-export or \$export on them and fetch what the export produced, and nothing else. Another provider's Group answers not-found rather than forbidden, so a token does not learn which other attribution lists exist; every other request, member matching and ordinary search included, is refused. The limit applies to API tokens; a token from a SMART issuer carries no Group limit. When SMART is configured as well, API tokens are refused unless **Settings → FHIR → Accept Perfuse API tokens beside SMART tokens** is on (see Security, *Who may call the FHIR endpoint*); a provider's Group-limited token needs it.

16.6 Prior authorisation metrics

perfuse cms0057 metrics reads a CSV of decisions exported from the utilisation management system and produces the public page CMS requires, in the layout of its *Prior Authorization Metrics Reporting — Overview and Template*.

```
perfuse cms0057 metrics -year 2025 -org "Springfield Health Plan" -services
services.csv \
    decisions.csv > prior-authorization-metrics-2025.html
```

Columns are found by header: priority, received, decided and decision are required, and request_id, line_of_business, extended, appealed, appeal_decision, service_code and service_description are read when present. A request belongs to the year it was decided in; one still pending is listed as excluded, never counted. Every row that cannot be read is reported with its line number.

The arithmetic follows the template: for standard and expedited requests separately, the share approved and denied, approved and denied within the decision timeframe, approved after the review time was extended, and approved after appeal — out of appeals, not out of requests — with the mean and median time from receipt to decision. A median under a day is written in hours rather than rounded to "0 days", which the template asks for explicitly. Partly approved requests are counted as approved, since the template has no category for them, and the page says how many there were. Metrics are reported per line of business, which is the level CMS requires.

The page is self-contained — no script, no external stylesheet or font — because it is posted on a public website by whoever maintains that, and anything it loads from elsewhere is something else that can break or track a visitor. -format csv and -format json give the same numbers for other uses. Without -services the page carries a notice that the list of items and services requiring prior authorisation is missing, because CMS requires it on the same page.

16.7 What was checked, and against what

The CARIN and PDex outputs were validated with the official HL7 FHIR validator, loaded with the published packages (hl7.fhir.us.carin-bb#2.2.0, hl7.fhir.us.davinci-pdex#2.2.0): professional, inpatient and outpatient claims and two PAS decisions, with no errors. The PDex inputs were the PAS guide's own examples rather than shapes written here to agree with the converter. The validator found five defects on the first run — among them that CARIN requires the profile version in meta.profile and the claim-type code system version on institutional claims — and each is fixed and covered by a test.

Building this found a defect older than it: the FHIR store dropped every element its internal model did not declare, so an ExplanationOfBenefit lost its items, diagnoses, adjudication and payment on the way in. The store now keeps whatever a resource carries, and a test loads a CARIN claim through the server and reads it back whole.

CRD's output - the updated order, and the CDS Hooks response against CRD's own logical model - and DTR's package output validate in the official HL7 validator against CRD 2.2.1 and DTR 2.2.0 with no errors.

DTR here is the payer's side of 2.2.0. `$questionnaire-package` takes the questionnaires named, those CRD named on an order, or the coverage assertion id CRD returned (context, while the same process is serving CRD), and returns each as a package bundle with a `QuestionnaireResponse` to fill in, the Libraries its CQL needs and their dependencies. The value sets its answers use travel in the package when they are loaded (`PUT /fhir/ValueSet/{id}`), and `$expand` lists their codes. `$log-questionnaire-errors` writes what a DTR app reports to the server log.

Adaptive questionnaires: write the whole questionnaire once, with SDC's `questionnaireAdaptive` extension and `enableWhen` on the questions that only apply sometimes. The package hands the app an empty shell, and each `$next-question` call adds the next question that applies to the answers so far, until there is nothing left to ask and the response is marked completed. The order and the branching are the payer's own rules; questions that do not apply are never shown.

Signing in as a client: DTR and PAS clients reach the payer with SMART Backend Services - a signed JWT exchanged for a token at the payer's authorization server. Point Perfuse at that server with `-smart-issuer` (and `-smart-jwks` if its address does not resolve from Perfuse), and pass `-smart-backend-services` so the discovery document says so. The Inferno DTR suite passes this way with Keycloak as the authorization server.

Perfuse can be that authorization server itself: `serve -fhir -smart-clients clients.yaml` issues Backend Services tokens at `/auth/token` to the clients listed, each with the public keys it signs its assertions with and the system scopes it may be granted, and the FHIR endpoint accepts them. No separate identity server is needed for a payer's machine-to-machine clients.

With `-smart-users users.yaml` as well, members and clinicians sign in there to authorize apps, which the Patient Access API needs: the app sends the person to `/auth/authorize`, they sign in, a clinician chooses the patient, and both see what the app asks for and can untick any of it. A member is only ever given their own record, whatever the app asks for. Each person in the file names the FHIR resource they are, `Patient/...` or `Practitioner/...`, and a password hash from `perfuse smart hash`; the password itself is never written down.

People can sign in at the organisation's own identity provider instead: `-smart-oidc oidc.yaml` names it (issuer, client id and where the secret is), and the sign-in page offers "Sign in with" it beside the password. Each person is linked by the provider's subject, `oidc_subject` in the users file, so the file still decides who may authorize apps and which FHIR resource they are; an account linked to nobody is refused. Codes, refresh tokens and revocations are kept in the database, so a restart signs nobody out.

Users → SMART apps in the console lists the registered apps and the people, and adds, edits and removes them. Each save is checked as the files are at start-up, written to them, and used at once. A

confidential app saved without a secret is given one, shown once; a password is set there and never shown back. Removing an app, or a person, stops the access tokens already issued to it or granted by them at once, not when they expire, and their refresh tokens with them.

Scopes may be granular, as SMART App Launch 2 and US Core define them: `patient/0bservation.rs?category=laboratory` reads only the Observations that search would find. A grant that is filtered for a type applies its filter to every search and every read of that type; an unfiltered grant for the type, or for *, lifts it. A filter that cannot be read grants nothing rather than everything.

16.8 Deciding prior authorisation requests: PAS

`serve -fhir -pas -crd-rules rules.yaml` answers Da Vinci PAS 2.2.1 at `POST /fhir/Claim/$submit` and `POST /fhir/Claim/$inquire`. The decisions come from the same rules file as CRD's answers, so the order a clinician is told needs authorisation is decided by the rule that said so:

```
- description: Physical therapy
  codes: ["PT"] # matched on the item's productOrService,
then the code of the order it requests
  system: https://codesystem.x12.org/005010/1365
  covered: covered
  pa: auth-needed
  pa_decision: approve # approve, deny or pend; empty: not-
covered is denied, no-auth approved, the rest pended
  details:
    - {code: allowed-quantity, value: "6"} # asked for more, it is certified
for six (A6, with the authorised detail)
  - description: MRI
    codes: ["62"]
    pa_decision: approve
    pa_alternative: {system: "https://codesystem.x12.org/005010/1365", code: "4"}
# approved as an X-ray instead (an added item)
  - description: Surgery
    codes: ["2"]
    pa: auth-needed # pended for a reviewer, asking for:
    questionnaire: https://payer.example.org/Questionnaire/surgery # a DTR
questionnaire
    pa_attachments: ["18776-5"] # and documents, by LOINC attachment code
```

Each item gets an X12 review action: A1 certified, A3 not certified, A4 pended, A6 modified, C cancelled (certification type 3). A line that names no service is not decided but answered with an error (X12 901 AG, correct and resubmit). The response echoes what a 278 response echoes — item trace numbers, the requested service dates, the encounter's dates, the transmission identifiers with sender and receiver swapped — and gives each line the payer's own administration reference number. A service no rule names is pended, never approved: a decision nobody wrote down is a decision for a person.

Pended requests wait for a reviewer, who answers with POST /fhir/Claim/\$decide:

```
{
  "resourceType": "Parameters",
  "parameter": [
    {
      "name": "claimResponse",
      "valueString": "<the ClaimResponse id>",
    },
    {
      "name": "decision",
      "valueCode": "approve",
    },
    {
      "name": "reason",
      "valueString": "Reviewed with the plan of care.",
    },
    {
      "name": "reviewer",
      "valueString": "<the reviewer's NPI>"
    }
  ]
}
```

decision may also be modify: certified, but not as asked, with quantity (valueDecimal, the units certified) and/or alternative (valueCoding, the service approved instead). The response then says A6 Modified, with the certified detail or the alternative as an added item, and an authorization number. item (valueInteger, repeatable) limits the decision to some lines. The decision becomes a new version of the response, marked as reviewed by a person, and with -fhir-subscriptions it is delivered on the PAS topic (PASSubscriptionTopic, filtered by org-identifier, the requesting organisation's identifier) as the full PAS Response Bundle, in the same transaction that records it. \$inquire returns the current response to every request that provider made about that member; another provider's inquiry finds nothing. \$decide needs a write scope; \$inquire is a read.

Requests and every version of their responses are kept in pas_requests and pas_responses in the server's database.

16.8.1 The reviewer queue

In the web console, **Exchange → CMS-0057 → Reviewer queue** lists the pended requests soonest due first: an expedited request (Claim.priority stat) is due 72 hours after it arrived, a standard one 7 days. Choosing one shows the member, the provider, each service with the units asked for and where it stands, what the response asked for, and every document received for it, with plain-text documents shown inline. An editor records the decision there: approve, deny, or modify (units certified, or another service approved instead), with a reason and the reviewer's NPI. A modification has to certify fewer units than were asked for; as many or more is an approval, and is refused as a modification. A service approved instead starts in the code system of the one asked for. It is the same decision \$decide makes, so the provider's subscription delivers it. Reading a request and deciding it are both written to the audit log.

Monitor → Team dashboards → Prior authorization shows, for the last 30 days, how many standard and expedited requests were decided within CMS-0057's timeframes, decided late, still waiting in time and overdue now, with the median time to decide, and how the services asked for were answered: approved, modified, denied or still pended. These two tiles are read from the requests themselves, so they are not in the dashboard's Grafana export, which says so in its description.

16.8.2 Receiving the documents a pended request asks for

A pended response asks for documents with a Task and one CommunicationRequest per document. The provider sends them with Da Vinci CDex \$submit-attachment (POST /fhir/\$submit-attachment): the TrackingId is the CommunicationRequest's identifier, the attachment control number (the Task's identifier is accepted too), AttachTo is preauthorization, and MemberId must be the request's member.

Each attachment's DocumentReference or QuestionnaireResponse is kept with the request and shown to its reviewer; a document arriving decides nothing.

16.8.3 Deciding in an X12 utilization management system

Most payers decide prior authorizations in a UM system that speaks X12 278. With `-pas-278-url`, Perfuse is the FHIR front of that system: each `$submit` is sent as one 278 request (005010X217), every item a patient event carrying its service type, procedure code, diagnoses, date and quantity, and the 278 response's action code and authorization number become the item's review action. Local rules are not consulted.

```
perfuse serve -fhir -pas -pas-278-url https://um.example/x12/278 -pas-278-  
receiver ACMEUM \  
-pas-278-token-env UM_TOKEN
```

A UM system that pends first and decides later posts its later 278 response to `POST /fhir/Claim/$decide-278`; each event is matched to its request by the trace number Perfuse sent, and decided as `$decide` would decide it. If the UM system cannot be reached, or answers with something that is not a 278 response, every item is pended for a reviewer and the provider is told only that; the reason is in the server log.

A subscription with `id-only` content tells the provider which answer changed without saying what it is: the notification's focus is `Bundle/<id>`. `GET /fhir/Bundle/<id>` returns the latest PAS Response Bundle, `GET /fhir/Bundle/<id>/_history/<n>` the version a notification sent, and `GET /fhir/ClaimResponse/<id>` the ClaimResponse alone. A token bound to one patient is told they do not exist, as for any record outside its patient.

Verified with the official Inferno PAS Server v2.2.1 suite (see verification). Not sent, though PAS marks them must-support: `ClaimResponse.adjudication` with a review action, `ClaimResponse.extension:authorizedProvider` and `ClaimResponse.item.extension:communicatedDiagnosis`, because the extensions they need are not allowed there by their own context in PAS 2.2.1 and the validator rejects a response carrying them; and `request.extension:DataAbsentReason`, because Perfuse always knows the request. Not implemented: forwarding a request to a payer's utilization management system as an X12 278, and attachments sent back with CDex.

A rules file is checked against CRD's own invariants when it loads: a questionnaire needs documentation, a service that is not covered cannot need authorisation, satisfied belongs to one case and not a rule, and anything conditional must give a reason.

Not verified: the exports and member matching against another payer's implementation, the metrics against a payer's published figures, and CRD against a real EHR's CDS Hooks client.

17 Security

Perfuse handles clinical data. This chapter is what it does about that and what it leaves to you.

17.1 Transport security

TLS on the web interface is configured with `-tls-cert` and `-tls-key`. Without them the interface is plain HTTP, which is acceptable on a loopback address for a first look and nowhere else — it carries credentials and message content, both readable on the wire.

TLS on channel transports is per source and per destination, under a `tls` block. The available settings are in the source and destination references.

`insecure_skip_verify` accepts any certificate. It exists because hospital systems routinely present certificates that will not verify, and refusing to connect would mean refusing to integrate. It is named to be conspicuous, and using it means the connection is encrypted but not authenticated — you have protection against passive interception and none against an interposed server.

An unverified TLS connection is not the same as a verified one, and the distinction is worth recording somewhere your auditors will find it. The better fix is almost always to obtain the far end's certificate and trust it explicitly rather than trusting everything.

17.2 Authentication and roles

Users are held in the database, with three roles:

Role	Can
Viewer	See channels, messages, statistics and traces.
Editor	Also create and change channels, and run comparisons.
Admin	Also manage users, settings and tenants.

The split that matters is Viewer and Editor. A great deal of useful work — investigating why a message failed, tracing it, searching the store — needs no ability to change anything, and giving somebody Editor so they can look at a message is how configuration gets changed by accident.

Note that Viewer can read message content, which is clinical data. It is not a low-privilege role in any sense that matters for privacy; it is a low-privilege role for **configuration**.

17.3 Who may call the FHIR endpoint

The FHIR endpoint (serve -fhir) checks one of two kinds of token:

- **Perfuse API tokens**, made under **Users → Machine credentials** or with perfuse token create. Each carries a role, and can be

limited to FHIR Groups. This is the default when no SMART authorization server is configured.

- **SMART access tokens**, from Perfuse's own authorization server (-smart-clients) or an outside one (-smart-issuer). Each

carries scopes, and the endpoint holds every request to them.

With SMART configured, only SMART tokens are accepted, so every caller is held to its scopes. An API token has a role and no scopes: a viewer token can read every patient's record. A payer who turned SMART on to limit each app to what it was granted should not have a way in that is not limited.

Some sites need both all the same - an internal claims feed that holds an API token and has no SMART client, beside member apps that sign in through SMART. For those, **Settings → FHIR → Accept Perfuse API tokens beside SMART tokens** (-fhir-api-tokens at start-up) lets API tokens in as well. It is off by default, it takes effect on the next request in either direction, and changing it is written to the audit log. A token is told apart by its shape: a SMART token is a signed JWT, and an API token never is, so one is never checked as the other. Where you can, give such a system a SMART Backend Services client instead, with only the system scopes it needs.

17.4 Audit

Configuration changes, logins, and access to message content are recorded with who, what and when.

The audit log is in the same database as the messages. That is convenient and it is a limitation: somebody who can modify the database can modify the audit log. If your requirements include tamper-evident audit, the log needs shipping somewhere append-only, and Perfuse's own log output is the way to do that.

17.5 Secrets in configuration

Passwords, passcodes and keys appear in channel configuration, which means they are in the YAML files.

Two consequences. The files need filesystem permissions that reflect what they contain, and if they are in version control — which is otherwise recommended — the repository holds credentials and must be treated accordingly.

Where a value is a secret, the interface renders it as a secret and the API does not return it once saved. That protects it from being read over somebody's shoulder or leaked in a screenshot. It does not protect the file.

17.6 What is in the database

The message store holds messages as received, which for a clinical feed means patient-identifiable data. It should be on encrypted storage, backed up as clinical data with the retention that implies, and access-controlled at the filesystem level as well as through Perfuse's own roles.

`cleanup.periodDays` limits how long it is kept. See the message store for the tension between that and having enough history for rhythm-aware alerting.

17.7 Tenants

A tenant is an isolation boundary: channels, messages, users and settings belong to one, and nothing crosses.

This exists for a service provider running feeds for several practices. Do not use it to separate departments within one organisation that need to see each other's traffic — the isolation is real and there is no cross-tenant view.

17.8 Test data

The generated messages described in testing exist partly as a privacy control. Reproducing a problem elsewhere, sending an example to a vendor, or populating a test environment are all things that otherwise get done with real messages.

The generator's output contains no patient data by construction rather than by redaction. Prefer it to hand-stripped real messages, which is a process that can be done incompletely and usually is.

17.9 Single sign-on

Three mechanisms, and local accounts alongside all of them.

OpenID Connect (`-oidc`), **a directory over LDAP** (`-ldap`), and **SAML 2.0** (`-saml`). Each is configured from **Administer** → **Sign-on** and written to the file the flag names, so the screen and the file are two views of one thing rather than two places to keep in step.

Local accounts are always available and cannot be turned off. That is deliberate: an on-premises integration engine whose only way in is an identity provider becomes unreachable exactly when the identity provider is unreachable, and a clinical interface is not a good place to learn that. The sign-on screen tells you how many local administrators exist before it encourages you to depend on anything else.

17.9.1 What a group mapping decides

All three map groups from the directory to Perfuse roles, and somebody in groups matching more than one gets the most privileged. That is the only safe direction to resolve it — the alternative is a person losing access they are entitled to because they are also in a lesser group.

A mapping that grants nobody anything is refused rather than saved. A configuration that authenticates people and turns all of them away looks identical to a broken product from the outside: the round trip

completes and everybody is denied with nothing explaining why.

Creating an account on first sign-in is off by default. With it off, somebody has to exist in Perfuse before they can sign in, so the directory decides who they are and Perfuse decides who is allowed in.

17.9.2 SAML, specifically

The parts worth knowing before you configure it:

- **Only the certificate you configure is trusted.** A certificate carried inside a response is never used. An attacker who can send a document can also put their own certificate in it, and a verifier that reads it is one anybody can authenticate to as anybody.
- **A response has to answer a sign-in this server started.** Perfuse remembers the request it sent and refuses a response naming anything else, or naming a request it has already answered. Without that check a valid response is a bearer token for whoever holds it: somebody signs in as themselves, keeps the response, and posts it into your browser, and you are then inside their account with the audit log recording their name.
- **Sign-ins started at the identity provider are therefore off by default.** Turn on **Accept sign-ins started at the identity provider** only if you need a portal tile, and knowing that it is what makes the above possible.
- **The group attribute has no default and is required.** Entra sends groups, Okta sends whatever the application was configured with, ADFS sends a claim URI. A guess would produce a sign-in that works and grants nobody anything. If a sign-in is refused for having no role, the server log names the attributes that did arrive — which is usually the whole answer.
- **The reply URL must match exactly** what is registered at the provider. A response says where it was destined and one addressed elsewhere is refused, because accepting it would mean accepting a response meant for a different service.

17.9.3 What each provider sends

The group attribute is where sign-ins go wrong, and the three common providers each name it differently.

Provider	Usually sends	Notes
Keycloak	groups	Needs a group membership mapper added to the client.
Entra ID	a claim URI ending /claims/groups	Sends group object ids unless the application is configured to send names. Map the ids, or change the provider.
Okta	whatever the application was configured with	Commonly groups, with a filter deciding which are sent.
ADFS	a claim URI	Signs the Response rather than the Assertion, which Perfuse accepts.

A claim URI may be entered by its final segment: typing groups finds

`http://schemas.microsoft.com/ws/2008/06/identity/claims/groups`. A name that merely ends with the word does not match — `excluded_groups` is not `groups`.

Group membership arriving as one comma-separated value is treated as a single group name and matches nothing. That is the correct reading of the standard, and splitting on commas would mean a group legitimately containing one silently became two. If that is what your provider sends, change it there.

Test on the sign-on screen checks everything that can be checked without a person: the settings, the certificate, and whether a sign-in request can be built. It says plainly that it cannot prove somebody can sign in, because that needs a real assertion about a real person and the only way to get one is for them to try.

17.10 What Perfuse does not do

It does not encrypt the database at rest. Use filesystem or volume encryption.

It does not manage its own certificates or renew them. Point it at files and manage those files with whatever you already use.

SAML has been verified against two independent providers, each with a real browser sign-in, a real assertion and a real session: Keycloak 26 and Microsoft Entra. Two matters more than twice as much as one, because the two disagree in a way that turned out to be load-bearing — Keycloak writes its assertion with namespace prefixes and Entra writes the same elements without them, and the one defect this area has had was in prefix handling.

Start by reading your provider's metadata. Settings, then Sign-on, then the SAML tab: give the metadata URL or paste the document, and the sign-on address and signing certificate are filled in for you. Every SAML provider publishes one of these — Okta, Entra, AWS IAM Identity Center, Keycloak, ADFS — and it is the difference between configuring a provider and transcribing one. Doing it by hand means finding the certificate inside the XML, stripping the line breaks out of the base64 and wrapping it in PEM headers, where a single stray space produces a signature error that says nothing about formatting.

What comes back is described rather than dumped: each certificate's subject, expiry and SHA-256 fingerprint, so you can check it against what your provider's own console shows. An expired certificate is called expired rather than left as a date to compare by eye. Nothing is saved by reading — applying it to the form is a separate click, because a document somebody pasted is not yet a decision about what to trust.

A metadata URL is fetched over https only, and the address is checked against the same policy that governs channel destinations, so this cannot be used to make the server read cloud instance credentials. Redirects are not followed, because a permitted address redirecting to a blocked one would walk straight past that check.

If you are configuring Entra, five things will not match what the examples show you. It sends no email claim at all. Its NameID is an opaque identifier rather than an address. It sends no groups claim unless the application is explicitly configured to emit one, so a role mapping written against groups

refuses every sign-in. When groups are emitted they are object GUIDs rather than names, unless your tenant synchronises from on-premises Active Directory. And its claim names live under `schemas.microsoft.com/identity` where most documentation shows `schemas.xmlsoap.org`. Perfuse handles all five; they are listed because the first four will otherwise look like faults in Perfuse when they are Entra being Entra.

That distinction is not theoretical here. The first genuine Keycloak assertion failed outright, because the canonical form the signature covers was being computed with namespace prefixes dropped. No real identity provider could have signed anybody in, and the package's own five canonicalisation tests passed throughout — each of them signed and verified with the same code, so they proved only that it agreed with itself. If you are the first to point a different provider at this, expect to find something, and the server log will name the stage that failed.

18 Operations

18.1 Deployment

One binary and two paths: a directory of channel files and a database file. There is no application server, no separate web tier and no message broker required.

```
perfuse serve -channels ./channels -db ./perfuse.db -addr 0.0.0.0:8443 \
  -tls-cert ./cert.pem -tls-key ./key.pem
```

Run it under whatever supervises services on the host — systemd, launchd, a container runtime. It expects to be restarted if it exits.

18.2 Configuration as files

Channels are files, which means they can be in version control, reviewed before deployment, and diffed after an incident.

Files are re-read without a restart. A file that does not parse is reported and the previously loaded version keeps running, so a syntax error cannot take a working feed off the air.

A channel edited in the web interface is written back to its file. If those files are in version control, expect commits from configuration changes made through the interface, and do not hand-edit a file while somebody has it open in the builder.

18.3 Backup

Two things to back up, and they have different characteristics.

The channel directory is small, changes rarely, and is what you need to rebuild the service. Back it up as configuration; version control is usually sufficient.

The database holds messages, users and audit history. It is large, changes constantly, and contains clinical data — so it needs backing up as clinical data, with the retention and encryption that implies. Losing it does not stop the service, but it loses the ability to investigate anything that already happened and the history the rhythm learner depends on.

18.4 Restart behaviour

On restart, channels are loaded and sources begin listening. Queued deliveries resume.

The thing to understand is what happens to work in flight. Under the default `on_delivery` acknowledgement, a message that had not yet been acknowledged is not acknowledged, so the sender

will resend it — which is the correct outcome. Under `on_receipt`, a message that was acknowledged but not yet delivered is in the queue, and resumes; if the queue is disabled, it is lost. See message flow.

18.5 Upgrading

Replace the binary and restart. The database schema is migrated forward automatically on start.

Migrations are forward-only. There is no downgrade, so a rollback to a previous binary after the schema has moved needs the previous database. Take a copy before upgrading — this is the one operational precaution worth being disciplined about.

18.6 Monitoring

Alerting from within Perfuse is covered in alerting and is what will tell you a feed has stopped.

From outside, monitor the process, the disk the database is on, and whether the web interface responds. The database grows and a full disk stops message recording, which is a failure mode with no other warning.

18.7 Capacity

Perfuse's throughput ceiling has not been measured on real hardware under real traffic, and this manual will not print a number it cannot support.

What can be said: the architecture is one goroutine per connection with per-destination queues, so throughput scales with destinations rather than being serialised through one worker; message recording is a SQLite write per message, which makes the disk the most likely first constraint; and the message store grows linearly with traffic, so retention is the setting that governs long-term disk use.

If you are sizing this for a large feed, measure it with your own traffic using the generated test corpus described in testing. That is the honest answer, and it is also a better answer than a number from somebody else's hardware.

18.8 Where this installation got stuck

Activity carries a panel, visible to administrators, showing what Perfuse has refused to do — grouped by message, most frequent first — and how far the installation has got towards a working channel.

It exists because every other claim about whether this software is easy to use comes from the people who wrote it. A refusal does not: somebody wanted something, the server said no, and neither party was guessing. It is the only evidence in the product that does not come from BioDream LLC.

The funnel names the step that has not happened yet rather than leaving timestamps to compare. The state worth watching for is **messages arrive and none has been delivered to a destination** — a channel can take traffic for weeks and deliver none of it, and from every other screen that looks like it is working.

Nothing anybody typed is recorded. The table holds the request route with identifiers replaced by {id}, the status, and the server's own sentence. A validation message names a field and a rule, which is the useful part; the value that broke the rule is very often patient data, and a table of those would be a worse liability than the friction it measured.

Two deliberate exclusions. A 5xx is a fault in this software and is logged as one already — mixing the two would bury the cases where somebody could not work out what to type under the cases where nothing they typed would have helped. And an expired session produces a run of 401s that say nothing about usability and would drown every finding that does.

Reading the report also trims the table to its most recent five thousand rows. Refusals are 4xx responses so it grows slowly, and reading is a natural moment to tidy: no timer and nothing that runs when nobody is looking.

GET /api/friction returns the same thing for anyone who would rather read it as JSON.

18.9 Logs

Perfuse logs to standard output, which is where a supervisor or container runtime will collect it.

Message-level detail goes to the message store rather than the log, deliberately. A log line per message on a busy feed is noise that hides the lines that matter, and the store is searchable in ways a log file is not.

The log carries what the store cannot: startup and shutdown, configuration load results, channel state changes, and errors that are not attributable to a specific message.

18.10 Running two instances

Two Perfuse processes against the same channel directory and the same database is not supported and will not behave well — both will try to listen on the same ports and both will write to the same SQLite file.

For availability, run one instance and make its restart fast. MLLP senders queue and retry, so a short outage is absorbed by the sender rather than losing data. This is a deliberate trade in favour of simplicity, and it means Perfuse is not the right choice if your requirement is continuous availability through a host failure.

18.11 More than one server

A site with two instances has a question no single console can answer: is everything running? Mirth charges for the answer. This is the fleet view.

One instance is nominated as the place you look. It polls the others and shows every channel on every server in one table, with a rollup across all of them.

18.11.1 Adding a peer

Settings → Fleet, or PUT /api/fleet/peers. A peer needs a name, a URL, and a token issued by the peer itself.

The name is required and is not the URL, because a URL is not something anybody recognises at three in the morning.

The token must be created **on the peer**, under Users, and should be viewer-scoped. Adding a peer without one is refused, with those instructions, rather than accepted and then failing quietly on every poll. Reading another instance's health is the smallest privilege there is, and it is the only one the fleet view needs.

18.11.2 Controlling a peer, which is off

`allow_control` permits starting and stopping that peer's channels from here. It is off by default and every use is audited by name.

The two privileges are deliberately separate. Reading another server's health is minor; stopping its channels during a transfusion is not, and the second should never arrive silently attached to the first.

18.11.3 What the report says

Each server reports the total number of channels, how many are running, stopped or errored, the queue depth and the age of the oldest waiting message, how many alerts are firing, and whether it is draining for shutdown.

The rollup counts servers three ways: **reachable**, **unreachable** and **undetermined**. The third covers peers whose health nobody knows — not yet polled, refusing the token, or running a version this one cannot read. It is kept apart from unreachable because "we cannot tell" is not "it is down", and folding the two together is how a fleet page starts lying.

For the same reason the rollup carries **knownFrom**: how many instances the channel and queue figures were actually read from. Twelve channels running across a fleet means something different when two of five servers did not answer, and the total alone cannot say so.

An unreachable peer says why in terms an operator can act on. A refused connection is reported as the host being up with nothing listening on that port, which is a different problem from a host that does not answer at all.

18.11.4 Clock skew

The rollup reports the largest difference between the clocks of the servers in it. Worth a number of its own: correlating an incident across two instances whose clocks disagree by four minutes produces a sequence of events that did not happen.

18.11.5 What a peer learns about you

Counts and rates. The report one instance gives another carries no message identifiers, no channel-level detail and no patient data of any kind.

That is what stops a fleet view being a centralisation of clinical data: the aggregating instance learns how many channels are running and nothing whatsoever about what flowed through them. If you want the messages you open that server's own console, where the audit log records that you did.

18.11.6 A peer pointed at itself

Refused. An instance polling its own address through its own HTTP stack appears twice in its own fleet view and double-counts every channel it has.

18.11.7 TLS on internal networks

`insecure_skip_verify` accepts a peer's certificate without verifying it. It exists because hospital infrastructure runs on private certificate authorities, and refusing to work at all would push people onto plain HTTP, which is worse. It is named so it cannot be mistaken for a good idea.

19 The Command Line

Everything Perfuse does to a running system is reachable from the web interface, and that is deliberate: a feature that needs a terminal is treated as a defect rather than a design choice. The commands here are for the work that happens *around* a running system — before it exists, while you are deciding whether to trust it, and when you need to hand something to somebody else.

Several of them never touch a server at all. `perfuse explain`, `perfuse profile`, `perfuse deident` and `perfuse translate` read files and write files, so they are useful on a laptop against an estate that still runs entirely on something else.

Run any command with `-h` for its flags. This chapter says what each one is *for*.

The flags shown here are illustrative, not exhaustive. `-h` is generated from the code and cannot fall out of date; a list copied into prose can.

19.1 Running channels

19.1.1 `perfuse serve`

Runs the web interface and the API, and by default runs the channels too. This is the command an installation uses.

It refuses to serve plain HTTP to anything except loopback unless you pass `-insecure`. That is not configurable politeness: clinical traffic and a session cookie over an unencrypted LAN connection is a reportable event, and the flag exists so that saying yes is a decision somebody made rather than a default they inherited. See security.

19.1.2 `perfuse run`

Runs channels from YAML with no web interface and no database. Useful in a container that should do exactly one thing, and useful when you want to see a channel's behaviour without anything else running.

19.1.3 `perfuse check`

Validates channel definitions and prints what they do, without running them. The second half is the point: a channel that loads is not necessarily the channel you meant, and reading a plain-English summary of what a file will do catches a mistake that validation cannot see.

Exits non-zero on a fault, so it belongs in whatever checks your configuration before it reaches a server. `-quiet` prints nothing on success, for that use.

`-allow-metadata-egress` permits destinations pointed at cloud instance metadata addresses. Those addresses hold the machine's own credentials, so a channel that can be talked into sending there is a credential leak; the check refuses them unless you say otherwise.

19.1.4 perfuse test

Runs tests written against channel definitions. A test sends a message through the channel's real filter, real transformations and real scripts, in the real order, with only the network replaced — so what passes here is what the channel does.

Tests live in `*_test.yaml` or `*.test.yaml` beside the channels. See testing.

19.2 Migrating from another engine

These three are the ones to run first, before deciding anything. See migration.

19.2.1 perfuse explain

Reads Mirth, Open Integration Engine and BridgeLink exports — a channel, a channel group or a whole server backup — and describes what each channel does and what would block a migration. It changes nothing and needs no Perfuse installation, which makes it the cheapest possible first step: point it at an export of your estate and read what comes back.

`-strict` exits non-zero if anything is blocked, so it can gate a pipeline. `-json` emits the same findings for a machine.

19.2.2 perfuse translate

Converts Mirth, Open Integration Engine and BridgeLink exports into Perfuse channel files: one channel, a channel group, a code template export or a whole server backup.

```
perfuse translate -o ./channels server-backup.xml
```

Code template libraries are written to `lib/` beside the channels, and each channel Mirth had a library enabled for includes it. See migration.

Nothing is silently dropped. Every part of a channel is either translated, carried across as a script that runs unchanged, or reported as needing a human. The third category is the honest one — a translator that produced a clean-looking file for every input would be hiding the decisions you most need to make.

19.2.3 perfuse compare

Runs the same traffic through two engines and reports where they disagree, grouped by field.

This is the command that answers "why should I trust this with patient data", and it answers it with evidence you gathered rather than a claim made here. It is the recommended last step before cutting over, and the recommended first step in any argument about whether Perfuse is ready.

19.3 Understanding a feed

19.3.1 perfuse profile

Reads messages and reports what is actually in them: which fields are always populated, which never are, the real value sets of coded fields, repetition counts, and which Z-segments appear.

Values are only reported for fields whose values form a small code set, so a profile of real traffic does not become a file full of patient data.

Saving a profile and comparing against it later is how you find out that a sender changed something:

```
perfuse profile -save last-month.json corpus/ perfuse profile -against last-month.json new/
```

19.3.2 perfuse contract

Turns the output of that thinking into something enforceable. A contract says what a feed must look like, so that the day it changes you are told, rather than finding out weeks later from a receiver that fell over — the messages are still valid HL7 when a field disappears, so nothing else notices.

```
perfuse contract promote -o adt.contract.yaml corpus/ perfuse contract check adt.contract.yaml today/
```

`promote` generates a starting point, not an answer. Read it and delete most of it: every line records whether it was measured or decided, which is what tells you which lines to keep.

19.4 Producing messages

19.4.1 perfuse generate

Produces synthetic HL7 v2 messages. Everything is invented — obviously fictional names, sequential identifiers, made-up addresses — so the output is safe to commit, mail, and attach to a ticket.

It can send straight at a channel rather than writing a file, which is the fastest way to prove a listener works:

```
perfuse generate -n 100 -send 127.0.0.1:6661
```

19.4.2 perfuse deident

Reads real messages and writes a corpus that keeps their structure, their codes and their intervals, and none of what identifies a patient. This is the difference between being able to send somebody a reproduction and not.

It is not the same tool as `perfuse generate` and the distinction matters. Generated messages are invented and therefore safe but unrealistic; a de-identified corpus is **derived from real traffic**, so it reproduces the oddities that actually break interfaces.

Two properties worth understanding before using it:

- The same salt always produces the same corpus, so a shared corpus can be regenerated rather than archived.
- **Keep the salt secret.** Pseudonyms are derived from real values, so anybody holding the salt can confirm a guess about who a record refers to.

`-keep-local-segments` passes Z-segments through unchanged and is unsafe. Z-segments are exactly where sites put names, notes and free text, so keeping them defeats the purpose of the tool. It exists because a structural problem is sometimes **in** a Z-segment, and then you need a corpus you must not share.

19.5 Moving messages by hand

19.5.1 **perfuse listen**

Accepts HL7 v2 over MLLP and acknowledges it. A receiver that exists for as long as you need it, for proving that a sender can reach this machine at all.

19.5.2 **perfuse send**

Sends HL7 v2 messages over MLLP from files.

MLLP has no authentication and no encryption, in Perfuse or anywhere else. Both of these commands are subject to that, and so is every MLLP source and destination: access is restricted at the network layer or it is not restricted.

19.6 FHIR

19.6.1 **perfuse fhir**

Converts HL7 v2 to FHIR, validates FHIR, or serves it, depending on the subcommand.

The validating subcommand is useful on its own: it will tell you whether a resource somebody sent you satisfies US Core, which is the profile American regulation is written against. Being able to answer that without adopting an engine is the point.

19.6.2 FHIR flags on perfuse serve

Flag	Effect
<code>-fhir</code>	Serve the FHIR endpoint at <code>/fhir</code> . On by default.
<code>-fhir-version</code>	The release served. R4 by default.
<code>-fhir-read-only</code>	Refuse writes.
<code>-fhir-api-tokens</code>	With SMART configured, accept Perfuse API tokens as well as SMART tokens (off by default; also under Settings → FHIR).
<code>-fhir-bulk-export</code>	Enable <code>\$export</code> . Off by default.
<code>-fhir-subscriptions</code>	Enable topic subscriptions: encounter and appointment notifications to endpoints FHIR clients choose. Off by default; needs R4 and a writable endpoint.
<code>-fhir-subscriptions-allow-http</code>	Let subscriptions deliver to plain http. For a test receiver, never for patient data on a network.

19.7 Setting up and operating

19.7.1 perfuse init

Creates the directory layout, an example channel, and a service definition for the operating system it is run on. Nothing existing is replaced unless you pass `-force`.

```
perfuse init -dir /opt/perfuse -service systemd
```

19.7.2 perfuse service

Registers Perfuse as a Windows service, removes it, or reports whether it is installed and running. Windows only; on Linux the equivalent is the unit file `perfuse init` can write for you. See operations.

19.7.3 perfuse token

Issues credentials for machines rather than people. A token does not expire and is not affected by anybody signing out, which is what a server polling its neighbour needs and what a browser session deliberately is not.

```
perfuse token create -db perfuse.db -label fleet-from-site-a -role viewer
perfuse token list -db perfuse.db
perfuse token revoke -db perfuse.db -label fleet-from-site-a
```

Only the hash is stored, so a token is displayed once when it is created and cannot be recovered afterwards. A stolen database therefore yields no usable token. Give a token the lowest role that does the job: a fleet view needs viewer, and nothing that only reads should hold anything more.

`-fhir-groups riverside-attributed,riverside-north` limits a token to those FHIR Groups, for a provider's CMS-0057 Provider Access export: on the FHIR endpoint it can read those Groups and export them, and nothing else. See CMS-0057 for payers.

19.7.4 **perfuse athena**

Prints the Amazon Athena `CREATE EXTERNAL TABLE` for an S3 archive written with format: `ndjson`, using partition projection on the `dt=YYYY-MM-DD` part of the key, so no crawler has to run as days are added.

`perfuse athena -bucket hospital-archive -destination archive`

Printed rather than run, because creating a table needs Athena and Glue permissions this program should not hold.

19.7.5 **perfuse sbom**

Lists everything linked into the binary, read from the build itself rather than from `go.mod` — so it describes the binary in front of you and not what somebody once asked for.

`-json` emits CycloneDX and `-spdx` emits SPDX tag-value, which is usually what a procurement or security review is actually asking for. The browser code embedded in the binary is included, because a reviewer matching a binary against advisories needs to know it contains a React application.

19.7.6 **perfuse version**

Prints the version and the commit it was built from. Worth putting in a ticket: "the current one" is not a version, and two machines that were installed a week apart are frequently not running the same build.

20 Channel Reference

Every key that can appear at the top level of a channel file. These are extracted from the source, so this chapter cannot describe an option that does not exist or omit one that does.

20.1 Channel

Channel is one configured message flow: somewhere messages arrive, an optional filter, and one or more places they go.

Defined as Channel in `internal/config`.

Key	Type	Required	Default
name	text	yes	—
description	text	no	—
group	text	no	—
attachments	Attachments (a block, omit to leave off)	no	—
delimited	Delimited (a block, omit to leave off)	no	—
tables	list of text	no	—
contract	ContractRef (a block, omit to leave off)	no	—
enabled	true or false (a block, omit to leave off)	no	—
dataType	DataType	no	hl7
hl7v3	HL7v3Options (a block, omit to leave off)	no	—
x12	X12Options (a block, omit to leave off)	no	—
ncpdp	NCPDPOptions (a block, omit to leave off)	no	—
script	ScriptOptions (a block, omit to leave off)	no	—
dicom	DICOMOptions (a block, omit to leave off)	no	—
source	Source	yes	—
filter	text	no	—
transformations	list of transform.Step	no	—
scripts	Scripts (a block, omit to leave off)	no	—
shadow	Shadow (a block, omit to leave off)	no	—
destinations	list of Destination	yes	—

20.1.1 name

Name identifies the channel in logs and metrics.

Required and unique.

20.1.2 **group**

Group is what this channel is part of, for organising a list that has grown too long to read.

Cosmetic at five channels and structural at two hundred: a site's channels divide by the system they talk to, or by the team that owns them, and a flat list of two hundred names is unusable however good the rest of the interface is.

A plain string rather than a group object with its own file, because a group has no behaviour. The moment it has a file it acquires settings, and settings on a group mean a channel's behaviour depends on something not written in the channel - which is the property that makes Mirth configuration hard to reason about. Groups here are a label and nothing more.

20.1.3 **attachments**

Attachments moves large payloads out of the message and puts them back before delivery.

Opt-in per channel: a feed of plain ADT messages has nothing to extract and should not pay for the machinery.

20.1.4 **delimited**

Delimited configures a delimited channel.

Only meaningful when the data type is delimited.

20.1.5 **tables**

Tables names files of shared mapping tables a transformation may refer to by name.

Shared because the domain tax is largely re-deriving a mapping somebody already derived: a site's sex codes and patient classes get translated identically in every channel that touches them, and a table inside a channel is a table only that channel can use.

20.1.6 **enabled**

Enabled defaults to true.

A disabled channel is loaded and validated but not started, so a file that is temporarily off is still checked by CI.

20.1.7 **dataType**

DataType is the format of the messages this channel receives.

Defaults to hl7, so every channel written before this existed keeps working.

20.1.8 **x12**

X12 holds the settings that only mean anything on an X12 channel.

Setting it on any other kind is refused rather than ignored.

20.1.9 **script**

Script configures an NCPDP SCRIPT channel.

Its steps live here rather than in the top-level transformations for the same reason every other non-v2 format's do: the step types differ per format, so one key would mean different things depending on `dataType`.

20.1.10 **dicom**

Imaging configures a DICOM channel's named transformation steps.

Named Imaging rather than DICOM because Source and Destination already have a DICOM field and three fields with the same name at different levels is how somebody sets the wrong one. The yaml key is `dicom`, which is what a person writing the file expects, and the Go name is the one that stops a mistake at the call site.

20.1.11 **filter**

Filter is an expression that must hold for a message to be forwarded.

A message that does not match is still acknowledged: the sender did nothing wrong, we simply are not interested.

20.1.12 **transformations**

Transformations are declarative changes applied to accepted messages, in order, before any script runs.

This is the layer meant to carry the ordinary work: each step is validated when the file loads, shows up in a diff, and can be reviewed by somebody who does not read JavaScript.

20.1.13 **scripts**

Scripts holds JavaScript for the cases the declarative steps cannot express, and is what a Mirth channel's own scripts drop into unchanged.

A channel with a script is marked as such in the interface, because its behaviour cannot be established by reading the configuration.

20.1.14 **shadow**

Shadow runs a candidate version of this channel beside it and reports where they differ.

Nothing it does can reach a receiver.

21 Source Reference

Every way a channel can receive messages, and every key each one accepts. A channel has exactly one source.

21.1 AMQP Source

AMQPSource receives from a queue or subscription.

A message is accepted only after the channel has handled it. One that could not be handled is released as a failed delivery, so the broker redelivers it and - on Service Bus, past the queue's max delivery count - dead-letters it.

Defined as AMQPSource in `internal/config`.

Key	Type	Required	Default
prefetch	whole number	no	10
reconnect	duration, such as 30s or 5m	no	5s

21.1.1 prefetch

Prefetch is how many messages the broker may send ahead.

Defaults to 10. They are still handled one at a time, in order.

21.1.2 reconnect

Reconnect is the wait after a lost connection.

Defaults to 5s.

21.2 Azure Blob Source

AzureBlobSource picks blobs up from a container prefix.

Defined as AzureBlobSource in `internal/config`.

Key	Type	Required	Default
prefix	text	yes	—
suffix	text	no	—
after_read	text	no	—
move_to	text	no	processed/
error_prefix	text	no	error/
poll_interval	duration, such as 30s or 5m	no	30s
max_object_size	whole number	no	16 MiB
framed	true or false	no	—

21.2.1 move_to

MoveTo is where a handled blob moves.

Defaults to processed/.

21.2.2 error_prefix

ErrorPrefix is where a blob that cannot be handled moves, so it does not block the rest.

Defaults to error/.

21.2.3 poll_interval

PollInterval between listings.

Defaults to 30s.

21.2.4 max_object_size

MaxObjectSize refuses a larger blob, which is left where it is.

Defaults to 16 MiB.

21.3 Broker Source

BrokerSource reads messages from a message broker.

Mirth's JMS Reader, reached over STOMP because JMS is a Java API rather than a protocol. See `internal/stomp` for why STOMP and not OpenWire.

Defined as BrokerSource in `internal/config`.

Key	Type	Required	Default
addr	text	yes	—
destination	text	yes	—
login	text	no	—
passcode	text	no	—
host	text	no	the address's host
selector	text	no	—
subscription_id	text	no	the channel name
heartbeat	duration, such as 30s or 5m	no	thirty seconds
reconnect	duration, such as 30s or 5m	no	five seconds
timeout	duration, such as 30s or 5m	no	thirty seconds
tls	tlsconf.Settings (a block, omit to leave off)	no	—

21.3.1 **addr**

Addr is the broker's host and port.

Required. 61613 is the usual STOMP port.

21.3.2 **destination**

Destination is the queue or topic to read from.

Required.

Brokers spell these differently - /queue/name on ActiveMQ, a bare name on RabbitMQ - so it is passed through unchanged rather than being assembled here. Guessing would mean working against one broker and silently reading nothing on another.

21.3.3 **login**

Login and Passcode authenticate.

Most brokers refuse an anonymous connection.

21.3.4 **passcode**

Login and Passcode authenticate.

Most brokers refuse an anonymous connection.

21.3.5 **host**

Host is the virtual host.

Defaults to the address's host.

21.3.6 **selector**

Selector filters messages at the broker.

A JMS selector, passed through rather than interpreted. Filtering at the broker rather than here means the messages this channel does not want never cross the network - which on a shared queue is the difference between reading a hundred messages a day and a hundred thousand.

21.3.7 **subscription_id**

SubscriptionID names the subscription.

Defaults to the channel name.

Named rather than generated, because a durable subscription is identified by it: a generated one would create a new subscription on every restart and leave the old ones accumulating messages nobody reads.

21.3.8 **heartbeat**

Heartbeat keeps the connection alive.

Defaults to thirty seconds.

On by default here, unlike the client's own default, because a source holds one connection open for weeks. Without heartbeats a connection through a firewall that has silently dropped the route looks perfectly healthy from this end and messages simply stop arriving, with nothing logged.

21.3.9 **reconnect**

Reconnect is how long to wait before reconnecting after a failure.

Defaults to five seconds.

21.3.10 **timeout**

Timeout bounds connecting.

Defaults to thirty seconds.

21.4 **DICOM Query Source**

DICOMQuerySource polls an archive with C-FIND and emits a message per new study.

Mirth has no equivalent connector. Its DICOM support is a listener and a sender - it can receive images pushed at it and push them on, and that is all. Asking an archive what it holds has been requested on their forums since 2015.

What makes it worth having is not the query, it is the automation: a prefetch that runs at three in the morning, a reconciliation that notices the archive is missing a study the RIS says exists, a worklist built from what was actually scanned rather than what was ordered. Those are jobs sites currently do with a scheduled script outside the engine, which means they are invisible to it - no retries, no alerting, no message history.

Defined as `DICOMQuerySource` in `internal/config`.

Key	Type	Required	Default
address	text	yes	—
called_ae	text	yes	—
calling_ae	text	no	—
level	text	no	STUDY
patient_root	true or false	no	—
match	mapping of text to text	no	—
return	list of text	no	—
interval	duration, such as 30s or 5m	yes	—
window	duration, such as 30s or 5m	no	—
overlap	duration, such as 30s or 5m	no	one hour
limit	whole number	no	500
timeout	duration, such as 30s or 5m	no	two minutes
tls	tlsconf.Settings (a block, omit to leave off)	no	—
emit_on_first_poll	true or false	no	false

21.4.1 address

Address is the archive's host and port.

Required.

Named to match the DICOM destination rather than shortened. The two were "address" and "addr" for an afternoon, which is the sort of inconsistency somebody hand-writing a channel file gets wrong once and then distrusts the whole format over.

21.4.2 **called_ae**

CalledAE is the archive's AE title.

Required in practice: most archives refuse an association addressed to anything else, and that refusal reads as an outage rather than as a configuration mistake.

21.4.3 **calling_ae**

CallingAE is the AE title Perfuse presents.

Archives commonly use it for access control.

21.4.4 **level**

Level is the query level: PATIENT, STUDY, SERIES or IMAGE.

Defaults to STUDY.

STUDY is the default because it is what almost every useful query wants and because the alternatives surprise people: IMAGE level against a large CT returns one response per slice, which is hundreds per study.

21.4.5 **match**

Match are the query keys.

A value filters; an empty value asks for the field to be returned.

Keys are named rather than numbered - "PatientID" not "0010,0020" - because a channel file is read by people, and a tag number in a configuration file is a lookup every reader has to perform.

21.4.6 **return**

Return are fields to bring back without filtering on them.

Separate from Match with an empty value, even though they encode identically, because the intent differs and a reader can see which keys are narrowing the search and which are being collected.

21.4.7 **interval**

Interval is how often to poll.

Required.

21.4.8 **window**

Window is how far back each poll looks, as a study date range.

Zero disables date filtering entirely.

A window is what keeps this bounded. Without one, every poll asks the archive for everything it has ever held and then discards what it has seen before - which works on a test archive and is antisocial against a real one holding millions of studies.

21.4.9 **overlap**

Overlap re-queries this far into the already-pollled period.

Defaults to one hour.

Necessary rather than cautious. A study can be registered with yesterday's date, our clock and the archive's need not agree, and a poll boundary that lines up exactly with an arrival loses it. The overlap re-asks, and the already-seen check stops the duplicate - so the cost of overlapping is a slightly larger query and the cost of not overlapping is a silently missed study.

21.4.10 **limit**

Limit caps the matches accepted from one poll.

Defaults to 500.

A cap rather than no cap because a mistyped match key turns this into "give me everything", and the first symptom of that against a real archive is the archive's administrator asking who is hammering it.

21.4.11 **timeout**

Timeout bounds one poll.

Defaults to two minutes.

21.4.12 **emit_on_first_poll**

EmitOnFirstPoll sends messages for everything the first poll finds.

Defaults to false.

False matters. Pointing this at an archive that already holds a million studies and having it emit a message for each is not a useful first run, and it is the kind of mistake that is noticed downstream rather than here. The first poll records what exists and emits nothing; the second poll onwards reports what is new.

21.5 **DICOM Source**

DICOMSource receives imaging objects as a C-STORE service class provider.

What a hospital points a modality or a PACS at. Mirth calls this the DICOM Listener and it does C-STORE only, so this is parity rather than a subset.

Defined as DICOMSource in `internal/config`.

Key	Type	Required	Default
listen	text	yes	—
ae_title	text	no	—
allowed_calling_ae	list of text	no	—
sop_classes	list of text	no	—
transfer_syntaxes	list of text	no	—
max_object_bytes	whole number	no	—
idle_timeout	duration, such as 30s or 5m	no	—
tls	tlsconf.Settings (a block, omit to leave off)	no	—

21.5.1 listen

Listen is the address to serve on.

Required. 104 is the registered DICOM port; 11112 is the common alternative where binding a privileged port is not possible.

21.5.2 ae_title

AETitle is what this endpoint calls itself.

Strongly recommended rather than required, and warned about when absent. A site's only access control on a DICOM endpoint is frequently the AE title, so an endpoint that accepts any title accepts anything that can reach it.

21.5.3 allowed_calling_ae

AllowedCallingAE lists the AE titles permitted to connect.

Empty accepts any caller.

This is the access control DICOM actually has. A PACS is routinely configured to accept one named calling title and reject everything else, and this end had no equivalent: the called title was checked - which is the name the caller dials, so anybody can send it correctly - while who was calling was recorded in the log and otherwise ignored.

Empty still accepts anything, because a modality nobody listed is the normal state of a first installation and refusing by default would break every deployment on day one. It warns instead, and the warning names the titles that have connected so somebody can fill this in from what actually arrived.

21.5.4 **sop_classes**

SOPClasses restricts what kinds of object to accept.

Empty accepts any.

Empty is a reasonable default here because this engine relays objects rather than interpreting them, so refusing an unfamiliar class would refuse valid images for no benefit.

21.5.5 **transfer_syntaxes**

TransferSyntaxes restricts the encodings to accept, in preference order.

Empty accepts the uncompressed three.

21.5.6 **max_object_bytes**

MaxObjectBytes bounds one received object.

Zero applies a default.

21.5.7 **idle_timeout**

IdleTimeout closes an association that has gone silent.

Zero means never, which is usually right - a modality holds an association open between studies.

21.5.8 **tls**

TLS encrypts inbound associations and can require a client certificate.

Worth stating plainly: Mirth needs its paid SSL Manager extension for this, and the community forks cannot ship it at all. DICOM carries patient names in its metadata, so this is one of the places where being a rewrite rather than a fork is a real advantage rather than a stylistic one.

21.6 **Database Source**

DatabaseSource reads rows from a database and turns each one into a message.

This is how a great many hospital interfaces actually start. A department system has no HL7 capability, so somebody writes rows into a staging table and the interface engine polls it. It is unglamorous and it is everywhere.

It is also the single commonest way a Mirth channel stalls, and the reason is almost always the same shape: a row that cannot be processed is read, fails, and is read again, forever, because nothing marked it. The channel looks alive. The queue looks empty. Nothing moves, and every message behind the bad row waits for a row that will never succeed.

Most of the design here is about that.

Defined as DatabaseSource in `internal/config`.

Key	Type	Required	Default
driver	text	yes	—
dsn	text	yes	—
query	text	yes	—
after_query	text	no	—
key_column	text	no	—
template	text	no	—
column	text	no	—
poll_interval	duration, such as 30s or 5m	no	—
batch_size	whole number	no	100
max_attempts	whole number	no	3
query_timeout	duration, such as 30s or 5m	no	30s

21.6.1 driver

Driver names the database.

See SupportedDrivers.

21.6.2 dsn

DSN is the connection string.

Prefer an environment variable reference over a literal, because this normally contains a password and channels are meant to live in git.

21.6.3 query

Query selects the rows waiting to be sent.

Required.

It must be a SELECT. A poll that quietly performed writes would run on a schedule with no record of what it changed.

21.6.4 after_query

AfterQuery marks a row as processed, and runs with the row's key bound to it.

Strongly recommended.

Without it, every poll re-reads every row the query still matches, so the query itself has to exclude processed rows some other way. If neither is true the channel resends the same rows forever, which is a duplicate storm rather than a stall and is arguably worse.

21.6.5 **key_column**

KeyColumn identifies a row for AfterQuery, quarantine and de-duplication.

Required when AfterQuery is set.

21.6.6 **template**

Template builds a message from a row.

Either this or Column is required.

Column references use \${column_name}. Missing columns are an error at load rather than an empty string in a clinical message.

21.6.7 **batch_size**

BatchSize caps the rows taken per poll.

Defaults to 100.

Bounded because an unbounded first poll against a table somebody has been filling for a year would read all of it into memory and then send all of it, which is how a migration floods a live receiver.

21.6.8 **max_attempts**

MaxAttempts is how many times a single row may fail before it is quarantined and the poll moves on.

Defaults to 3.

This is the setting that prevents the stall. Retrying forever is not resilience; it is one bad row stopping a hospital feed.

21.6.9 **query_timeout**

QueryTimeout bounds a single poll.

Defaults to 30s.

21.7 **FTP Source**

FTPSource collects files from an FTP or FTPS server.

Why FTP is still here

It is thirty years past its recommended replacement and a great many hospital systems still export to it: appliances whose firmware will never be updated, billing systems whose vendor charges for a change. Refusing to speak it does not make those systems go away, it means the integration engine cannot be used.

FTPS is the default. It is the same protocol with TLS, most surviving servers support it, and it is a much easier conversation with a security team than plain FTP.

Defined as FTPSource in `internal/config`.

Key	Type	Required	Default
host	text	yes	—
user	text	no	—
password	text	no	—
security	ftpconn.Security	no	explicit
insecure_skip_verify	true or false	no	—
root	text	no	—
timeout	duration, such as 30s or 5m	no	thirty seconds

21.7.1 host

Host is the server, with an optional port.

Port 21 is assumed.

21.7.2 user

User and Password authenticate.

Both may be left out for an anonymous server.

Anonymous FTP is still how some public reference feeds are published, so an empty user is a legitimate configuration rather than an oversight and is not refused. On a server that does expect credentials, omitting them produces a login failure at connect time, which is reported against the channel rather than retried silently.

21.7.3 password

User and Password authenticate.

Both may be left out for an anonymous server.

Anonymous FTP is still how some public reference feeds are published, so an empty user is a legitimate configuration rather than an oversight and is not refused. On a server that does expect credentials, omitting them produces a login failure at connect time, which is reported against the channel rather than retried silently.

21.7.4 security

Security is explicit, implicit or none.

Defaults to explicit, which is FTPS negotiated with AUTH TLS on the ordinary port. "none" is plain FTP and has to be asked for by name.

21.7.5 root

Root is the directory every path is relative to, and what containment is enforced against.

The server has no notion of it: an FTP server will happily accept ../.. as a path. So this is what stops a move_to from writing wherever the account can reach.

21.7.6 timeout

Timeout bounds one operation, not the whole poll.

Defaults to thirty seconds.

Per operation because a poll that lists a directory and fetches twenty files does twenty-one operations, and a single budget for all of them would abort a perfectly healthy transfer of a large file.

21.8 File Source

FileSource reads files from a directory this server can see.

The connector a site tries first

An analyser writes results to a folder. A billing system drops a batch overnight. Radiology exports reports to a directory somebody mounted years ago. None of it involves a protocol, and an integration engine that cannot read a folder fails its evaluation on the first afternoon.

Defined as FileSource in internal/config.

Key	Type	Required	Default
root	text	yes	—
follow_symlinks	true or false	no	—

21.8.1 root

Root bounds every path this source touches.

Required.

Separate from Dir so that move_to and error_dir can be relative and still be contained. Given only Dir, an error_dir of "../failed" would be outside anything the source had been granted, and refusing it would be arbitrary because there would be nothing to refuse it against.

21.9 GCS Source

GCSSource picks objects up from a bucket prefix, as the s3 and azure_blob sources do.

Defined as GCSSource in internal/config.

Key	Type	Required	Default
prefix	text	yes	—
suffix	text	no	—
after_read	text	no	—
move_to	text	no	processed/
error_prefix	text	no	error/
poll_interval	duration, such as 30s or 5m	no	30s
max_object_size	whole number	no	16 MiB
framed	true or false	no	—

21.9.1 move_to

MoveTo is where a handled object goes when after_read is move.

Defaults to processed/.

21.9.2 error_prefix

ErrorPrefix is where an object that could not be handled goes.

Defaults to error/.

21.9.3 poll_interval

PollInterval is how often the prefix is listed.

Defaults to 30s.

21.9.4 max_object_size

MaxObjectSize leaves larger objects where they are.

Defaults to 16 MiB.

21.10 HTTP Source

HTTPSource accepts messages over HTTP.

A listener rather than a poller: something posts a message and gets an acknowledgement back, which is how the modern half of the hospital integrates when MLLP is not available to it.

Defined as HTTPSource in `internal/config`.

Key	Type	Required	Default
listen	text	yes	—
path	text	no	"/"
tls	tlsconf.Settings (a block, omit to leave off)	no	—
token	text	no	—
max_message_size	whole number	no	—
read_timeout	duration, such as 30s or 5m	no	—
ack	Ack	no	—

21.10.1 listen

Listen is the address to serve on.

Required.

21.10.2 path

Path is the URL path to accept messages on.

Defaults to "/".

21.10.3 token

Token, when set, is required in an Authorization: Bearer header.

There is no default and no anonymous mode toggle: an HTTP endpoint on a hospital network with no authentication at all is a decision that has to be visible in the file, so leaving this empty is reported as a warning at load.

21.10.4 **max_message_size**

MaxMessageSize bounds an inbound body.

Zero applies a default.

21.11 **IBMMQ Source**

IBMMQSource reads a queue.

Defined as IBMMQSource in `internal/config`.

Key	Type	Required	Default
<code>poll_interval</code>	duration, such as 30s or 5m	no	2s

21.11.1 **poll_interval**

PollInterval is the wait when the queue is empty, or after a message that could not be handled.

Defaults to 2s.

21.12 **Java Script Source**

JavaScriptSource configures a source that runs a script on a timer and feeds the returned messages into the channel.

This is the Mirth "JavaScript Reader" equivalent.

The script runs in a goja VM with the same sandbox as transformer scripts: `msg` is not available (there is no inbound message yet), but `globalMap`, `configurationMap`, `logger`, and `DateUtil` are present. The script must return a string (one message) or an array of strings (many). Returning nothing or an empty array means the poll produced no messages, which is normal.

Defined as JavaScriptSource in `internal/config`.

Key	Type	Required	Default
<code>script</code>	text	yes	—
<code>poll_interval</code>	duration, such as 30s or 5m	no	—
<code>timeout</code>	duration, such as 30s or 5m	no	—

21.12.1 **poll_interval**

PollInterval is how often the script runs.

Zero means 5 seconds.

21.12.2 timeout

Timeout bounds a single script execution.

Zero means 30 seconds.

21.13 Kafka Source

KafkaSource reads messages from a Kafka topic.

Defined as KafkaSource in internal/config.

Key	Type	Required	Default
brokers	list of text	yes	—
topics	list of text	yes	—
group	text	yes	—
from_beginning	true or false	no	—
commit_after_delivery	true or false (a block, omit to leave off)	no	—
max_message_size	whole number	no	the channel's limit
session_timeout	duration, such as 30s or 5m	no	forty-five seconds
timeout	duration, such as 30s or 5m	no	thirty seconds
sasl	KafkaSASL (a block, omit to leave off)	no	—
tls	tlsconf.Settings (a block, omit to leave off)	no	—

21.13.1 brokers

Brokers is the bootstrap server list.

Required.

A list rather than one address, because a single bootstrap broker is a single point of failure for starting up: the cluster survives losing it and this channel would not.

21.13.2 topics

Topics are the topics to read.

At least one is required.

21.13.3 **group**

Group is the consumer group.

Required.

Named rather than generated, and required rather than defaulted, because it is what remembers how far this channel has read. A generated group would start from the beginning - or from the end, losing everything - on every restart, and it would leave the old group behind holding committed offsets nobody reads. Requiring it makes the person name the thing that has to stay the same across restarts.

21.13.4 **from_beginning**

FromBeginning reads the topic from its start when the group has no committed offset.

Off by default, so a new channel pointed at a topic with two years of history does not replay two years of patient events into a live system on the day it is switched on. The first person to hit that would be doing it in production, because that is where the topic with the history is.

21.13.5 **commit_after_delivery**

CommitAfterDelivery commits the offset only once the message has been handled.

On by default, and the reason is the whole difference between losing a message and seeing it twice. Committing on read means a crash between the commit and the delivery loses the message silently, and nothing anywhere records that it existed. Committing after means a crash in the same window redelivers it, which is visible and which HL7 receivers are built to tolerate - a duplicate A08 is a nuisance and a missing lab result is a patient safety event.

21.13.6 **max_message_size**

MaxMessageSize bounds one record.

Defaults to the channel's limit.

21.13.7 **session_timeout**

SessionTimeout is how long the group coordinator waits before assuming this consumer is gone and reassigning its partitions.

Defaults to forty-five seconds.

21.13.8 **timeout**

Timeout bounds connecting and metadata requests.

Defaults to thirty seconds.

21.14 **MQTT Source**

MQTTSource subscribes to a topic filter.

Defined as MQTTSource in internal/config.

Key	Type	Required	Default
topic	text	yes	—
reconnect	duration, such as 30s or 5m	no	5s

21.14.1 reconnect

Reconnect is the wait after a lost connection, or after a message that could not be handled.

Defaults to 5s.

21.15 Pub Sub Source

PubSubSource pulls from a Pub/Sub subscription.

Defined as PubSubSource in internal/config.

Key	Type	Required	Default
subscription	text	yes	—
max_messages	whole number	no	10
reconnect	duration, such as 30s or 5m	no	5s

21.15.1 max_messages

MaxMessages is how many are pulled at once.

They are still handled one at a time, in order. Defaults to 10.

21.15.2 reconnect

Reconnect is the wait after an error.

Defaults to 5s.

21.16 S3 Source

S3Source picks objects up from a bucket.

Defined as S3Source in internal/config.

Key	Type	Required	Default
bucket	text	yes	—
path_style	true or false	no	—
prefix	text	no	—
suffix	text	no	—
after_read	text	no	—
move_to	text	no	processed/
error_prefix	text	no	error/
poll_interval	duration, such as 30s or 5m	no	30s
max_object_size	whole number	no	16 MiB
framed	true or false	no	—

21.16.1 prefix

Prefix limits which objects are read, e.g. inbound/.

Empty reads the whole bucket, which is refused with after_read: move unless move_to is outside it - otherwise moved objects would be read again.

21.16.2 after_read

AfterRead is move (the default) or delete.

There is no leave: a bucket has no file a poller can remember having seen across a restart, so leaving objects means reading them all again every time Perfuse starts.

21.16.3 move_to

MoveTo is the prefix a handled object moves to.

Defaults to processed/.

21.16.4 error_prefix

ErrorPrefix is where an object that could not be handled moves, so one bad file does not block the rest.

Defaults to error/.

21.16.5 poll_interval

PollInterval between listings.

Defaults to 30s.

21.16.6 **max_object_size**

MaxObjectSize refuses anything larger.

Defaults to 16 MiB.

21.16.7 **framed**

Framed reads several MLLP-framed messages from one object.

Otherwise an object is one message.

21.17 **SFTP Source**

SFTPSource collects messages from a directory on an SFTP server.

Extremely common and rarely spoken about. A laboratory or a radiology system writes a file every few minutes, an SFTP server holds it, and something has to come and get it. Half the feeds described as "we send you HL7" are this.

The part worth care is not the transfer. It is that a file being written to and a file finished being written to look identical over SFTP, so a poll that is even slightly too eager collects half a message. That produces a truncated but often parseable HL7 message, which is the worst outcome available: it is accepted, acknowledged, stored and delivered, and the missing half is never mentioned again.

Defined as SFTPSource in `internal/config`.

Key	Type	Required	Default
host	text	yes	22
user	text	yes	—
password	text	no	—
key_file	text	no	—
key_passphrase	text	no	—
known_hosts_file	text	no	—
insecure_skip_host_key_check	true or false	no	—
dir	text	yes	—
pattern	text	no	everything
poll_interval	duration, such as 30s or 5m	no	—
after_read	text	no	—
move_to	text	no	—
error_dir	text	no	—
stable_for	duration, such as 30s or 5m	no	5s
framed	true or false	no	—
max_file_size	whole number	no	64MB
timeout	duration, such as 30s or 5m	no	60s

21.17.1 host

Host is the server, with an optional port.

Defaults to 22.

21.17.2 password

Password authenticates with a password.

Prefer KeyFile.

21.17.3 **key_file**

KeyFile is a private key.

The better choice, and the one hospital security teams ask for.

21.17.4 **known_hosts_file**

KnownHostsFile verifies the server's identity.

Required unless InsecureSkipHostKeyCheck is set. Without it there is nothing to distinguish the real server from anything that answers on that address, and the credentials are handed over before anybody notices.

21.17.5 **insecure_skip_host_key_check**

InsecureSkipHostKeyCheck accepts any host key.

Exists because refusing outright sends people to a shell script with StrictHostKeyChecking=no, which is worse in every way including auditability. It warns loudly and it is never the default.

21.17.6 **pattern**

Pattern selects files by glob.

Defaults to everything.

21.17.7 **move_to**

MoveTo is the remote directory files are moved into after reading.

Required when AfterRead is "move".

21.17.8 **stable_for**

StableFor is how long a file's size and modification time must be unchanged before it is read.

Defaults to 5s.

This is the setting that stops half a message being collected. Zero means reading whatever is there, which works right up until a file is large enough or a network slow enough that writing takes longer than a poll.

21.17.9 **framed**

Framed says the file contains MLLP-framed messages.

When false, the file is split on MSH boundaries.

21.17.10 **max_file_size**

MaxFileSize refuses a file larger than this many bytes.

Defaults to 64MB.

21.17.11 **timeout**

Timeout bounds a single connection.

Defaults to 60s.

21.18 **SMB Source**

SMBSrc collects files from a Windows file share.

Defined as SMBSrc in internal/config.

Key	Type	Required	Default
host	text	yes	—
share	text	yes	—
user	text	no	—
password	text	no	—
domain	text	no	—
root	text	no	—
timeout	duration, such as 30s or 5m	no	thirty seconds

21.18.1 **host**

Host is the server, with an optional port.

Port 445 is assumed.

21.18.2 **share**

Share is the share name only: the "data" in \\server\data.

A path here is refused rather than trimmed.

21.18.3 **user**

User and Password are the Windows account.

Required in practice.

A share reachable with no credentials at all is rare enough on a domain that an empty user here is far more likely to be a mistake than a guest share - but it is accepted, because refusing it would block the one arrangement where it is correct. Where the account belongs to a domain, set Domain as well: authenticating a domain account without it fails in a way that reads like a wrong password.

21.18.4 **password**

User and Password are the Windows account.

Required in practice.

A share reachable with no credentials at all is rare enough on a domain that an empty user here is far more likely to be a mistake than a guest share - but it is accepted, because refusing it would block the one arrangement where it is correct. Where the account belongs to a domain, set Domain as well: authenticating a domain account without it fails in a way that reads like a wrong password.

21.18.5 **domain**

Domain is the Windows domain or workgroup.

Empty is usually right for a local account.

21.18.6 **timeout**

Timeout bounds one operation.

Defaults to thirty seconds.

Worth raising on a share reached across a site link. An SMB operation against a distant server is slower than the same operation locally by enough that a default tuned for a LAN produces timeouts that look like the share being unavailable.

21.19 **SOAP Source**

SOAPSource accepts messages wrapped in a SOAP envelope.

The counterpart to the SOAP destination, and the rarer direction: most hospital integration has Perfuse calling a service rather than being one. It exists because Mirth's Web Service Listener does, and a site whose sending system only speaks SOAP cannot migrate without it.

Deliberately not a general web services framework. It accepts an envelope, takes the message out of it, and answers with an envelope. There is no WS-Security, no MTOM, no WS-Addressing, and no generated bindings - the same omissions as the destination, for the same reason: a half-implementation of WS-Security is worse than none.

Defined as SOAPSource in `internal/config`.

Key	Type	Required	Default
listen	text	yes	—
path	text	no	"/"
version	text	no	1
element	text	no	—
base64	true or false	no	—
response_element	text	no	"AckResponse"
response_namespace	text	no	—
wsdl	text	no	—
tls	tlsconf.Settings (a block, omit to leave off)	no	—
token	text	no	—
username	text	no	—
password	text	no	—
max_message_size	whole number	no	—
read_timeout	duration, such as 30s or 5m	no	—
ack	Ack	no	—
fault_on_nak	true or false	no	—

21.19.1 listen

Listen is the address to serve on.

Required.

21.19.2 path

Path is the URL path to accept envelopes on.

Defaults to "/".

21.19.3 version

Version is the SOAP version to answer in, "1.1" or "1.2".

Defaults to 1.1.

Requests are accepted in either version regardless, because a sender that gets this wrong is common and rejecting it teaches nobody anything. The setting controls the reply, where getting it wrong means the sender cannot read the answer.

21.19.4 **element**

Element names the element inside the SOAP Body holding the message.

Empty means the whole body text.

Named rather than guessed because a body may carry several elements - a message alongside a sender identifier and a timestamp - and taking the whole body would hand the engine an XML fragment where it expected HL7.

21.19.5 **base64**

Base64 decodes the element's content before treating it as a message.

Common in real services, because HL7 carries carriage returns and callers who did not think about that end up with XML that mangles them. If the content decodes as base64 but was not meant to be, the result is not a message and the engine says so - which is why this is explicit rather than sniffed.

21.19.6 **response_element**

ResponseElement names the element wrapping the acknowledgement in the reply.

Defaults to "AckResponse".

21.19.7 **wSDL**

WSDL is a file served at the listen path with ?wSDL.

Empty means none is served.

A file rather than something generated. A generated WSDL would describe what this endpoint accepts, which is almost never what the sending system was built against - and a WSDL that is nearly right costs more time than no WSDL at all, because it looks authoritative.

21.19.8 **username**

Username and Password, when set, require HTTP basic authentication.

Offered alongside a token because SOAP callers overwhelmingly expect basic authentication, and a site whose sending system cannot be changed needs the option it already has.

21.19.9 **password**

Username and Password, when set, require HTTP basic authentication.

Offered alongside a token because SOAP callers overwhelmingly expect basic authentication, and a site whose sending system cannot be changed needs the option it already has.

21.19.10 **max_message_size**

MaxMessageSize bounds an inbound body.

Zero applies a default.

21.19.11 **fault_on_nak**

FaultOnNak answers a negative acknowledgement with a SOAP fault instead of a normal response carrying the NAK.

Off by default, which is the right default and worth explaining. HL7 already has a way to say "I received this and rejected it", and that is a NAK - a successful exchange reporting a rejected message. Turning the same thing into a transport fault tells the sender its request was malformed, which it was not, and many clients respond by retrying forever.

It exists because some callers only look at the HTTP status and would otherwise treat a rejection as success.

21.20 **SQS Source**

SQSSource reads from an SQS queue.

A message is deleted only after the channel has handled it - delivered, or filtered, or failed in a way the channel records. Until then it stays on the queue, invisible for visibility_timeout, and comes back if Perfuse stops. So a message is never lost; the price is that one can arrive twice, which SQS itself already allows.

Defined as SQSSource in internal/config.

Key	Type	Required	Default
queue_url	text	yes	—
wait_seconds	whole number	no	20: an empty queue costs one request every twenty seconds
max_messages	whole number	no	10
visibility_timeout	duration, such as 30s or 5m	no	the queue's own setting

21.20.1 **visibility_timeout**

VisibilityTimeout is how long a received message stays hidden while it is handled.

Defaults to the queue's own setting.

21.21 Serial Source

SerialSource reads messages from a serial port.

Why a serial connector in 2026

Because the equipment is still there. A blood gas analyser bought in 2009 with a working sensor and a 25-pin socket, a bedside monitor, a scale in a dialysis unit, an older anaesthesia machine. None of them will ever get an Ethernet port, several of them cost six figures, and the alternative to reading them is somebody typing results into a form.

Mirth has a serial connector. Perfuse had nothing, and a site with one of these had no route in at all.

Framing is shared with the TCP connector

The framings are the same wherever the bytes come from: an analyser that speaks STX/ETX over a socket speaks STX/ETX over a cable. So this reuses the TCP framing block rather than defining its own, which also means a device moved from a serial cable to a serial-to-Ethernet adapter keeps its framing configuration.

What is different about a serial line

There is no connection. A socket tells you when the peer went away; a cable does not. An unplugged cable, a device switched off, and a device with nothing to say are indistinguishable - all three are silence. That is why `quiet_after` exists: without something to say "this line has been silent for longer than it should be", a dead feed looks exactly like a quiet night.

Defined as `SerialSource` in `internal/config`.

Key	Type	Required	Default
port	text	yes	—
baud	whole number	yes	—
data_bits	whole number	no	8
parity	text	no	none
stop_bits	text	no	1
flow_control	text	no	none
max_message_size	whole number	no	—
quiet_after	duration, such as 30s or 5m	no	—
reply	text	no	—
reply_text	text	no	—
reopen_after	duration, such as 30s or 5m	no	—

21.21.1 **baud**

Baud is the speed.

Required, because guessing produces bytes rather than an error.

A wrong baud rate does not fail. It delivers plausible-looking rubbish: framing bytes appear at random, occasional runs decode as printable characters, and a lenient parser accepts some of it. There is deliberately no default.

21.21.2 **parity**

Parity is none, odd, even, mark or space.

Defaults to none.

21.21.3 **stop_bits**

StopBits is 1, 1.5 or 2, written as "1", "1.5" or "2".

Defaults to 1.

21.21.4 **flow_control**

FlowControl is none, hardware or software.

Defaults to none.

Worth setting correctly rather than leaving. With hardware flow control expected and not configured, a device stops sending partway through a long message and the result is a truncated message rather than a failure.

21.21.5 **quiet_after**

QuietAfter logs a warning when nothing has been received for this long.

The only way to notice a dead serial feed. An unplugged cable and a quiet night are the same silence, so somebody has to say how long is too long for this particular device.

21.21.6 **reopen_after**

ReopenAfter is how long to wait before reopening a port that failed.

A USB serial adapter unplugged and plugged back in comes back as the same device path but a different kernel handle, so the old one returns errors forever. Reopening is the only recovery, and doing it in a tight loop fills the log.

21.22 **Source**

Source is where a channel receives messages.

Defined as Source in `internal/config`.

Key	Type	Required	Default
type	SourceType	yes	—
listen	text	yes	—
max_message_size	whole number	no	—
idle_timeout	duration, such as 30s or 5m	no	—
max_connections	whole number	no	—
tls	tlsconf.Settings (a block, omit to leave off)	no	—
http	HTTPSource (a block, omit to leave off)	no	—
soap	SOAPSource (a block, omit to leave off)	no	—
dicom	DICOMSource (a block, omit to leave off)	no	—
file	FileSource (a block, omit to leave off)	no	—
tcp	TCPSource (a block, omit to leave off)	no	—
ftp	FTPSource (a block, omit to leave off)	no	—
smb	SMBSrc (a block, omit to leave off)	no	—
webdav	WebDAVSource (a block, omit to leave off)	no	—
serial	SerialSource (a block, omit to leave off)	no	—
dicom_query	DICOMQuerySource (a block, omit to leave off)	no	—
broker	BrokerSource (a block, omit to leave off)	no	—
kafka	KafkaSource (a block, omit to leave off)	no	—
sqs	SQSSrc (a block, omit to leave off)	no	—
s3	S3Source (a block, omit to leave off)	no	—
amqp	AMQPSource (a block, omit to leave off)	no	—
azure_blob	AzureBlobSource (a block, omit to leave off)	no	—
mqtt	MQTTSource (a block, omit to leave off)	no	—

Key	Type	Required	Default
ibm_mq	IBMMQSource (a block, omit to leave off)	no	—
pubsub	PubSubSource (a block, omit to leave off)	no	—
gcs	GCSSource (a block, omit to leave off)	no	—
database	DatabaseSource (a block, omit to leave off)	no	—
sftp	SFTPSource (a block, omit to leave off)	no	—
javascript	JavaScriptSource (a block, omit to leave off)	no	—
ack	Ack	no	—

21.22.1 type

Type is the transport.

Only mllp is implemented.

21.22.2 max_message_size

MaxMessageSize bounds one inbound message.

Zero uses the transport default.

21.22.3 idle_timeout

IdleTimeout closes a connection that has been silent this long.

Zero means never, which is usually right: a hospital feed holds a connection open for months and goes quiet overnight.

21.22.4 max_connections

MaxConnections bounds concurrent connections.

Zero means unlimited.

21.23 TCP Source

TCPSource listens on a socket for messages that are not necessarily HL7.

Why this is separate from an mllp source

MLLP is one framing and Perfuse has always spoken it. A great deal of equipment does not: a laboratory analyser, a scale, a bedside monitor, an older billing system. Each tends to have its own convention, and

Mirth's TCP connector covers all of them, so a site with a socket feed that was not HL7 could not use Perfuse at all.

Defined as TCPSource in internal/config.

Key	Type	Required	Default
listen	text	yes	—
max_message_size	whole number	no	—
idle_timeout	duration, such as 30s or 5m	no	—
max_connections	whole number	no	—
reply	text	no	—
reply_text	text	no	—
tls	tlsconf.Settings (a block, omit to leave off)	no	—

21.23.1 idle_timeout

IdleTimeout closes a connection that has been silent this long.

Zero means never, which is usually right: a device feed holds a connection open for months and goes quiet overnight.

21.23.2 max_connections

MaxConnections bounds concurrent connections.

Zero means unlimited.

21.23.3 reply

Reply is what to send back after each message: nothing, ack, or a fixed string.

Most raw feeds expect nothing. Some expect a single ACK byte. A device that expects a reply and does not get one usually retries the same message forever, so this is worth being explicit about.

21.24 Web DAV Source

WebDAVSource collects files from a WebDAV collection.

Where this turns up: document management systems, SharePoint, Nextcloud, and vendor portals that expose a drop folder over HTTPS because it is the only outbound port their customers' firewalls allow.

Defined as WebDAVSource in internal/config.

Key	Type	Required	Default
url	text	yes	—
user	text	no	—
password	text	no	—
insecure_skip_verify	true or false	no	—
timeout	duration, such as 30s or 5m	no	sixty seconds

21.24.1 user

User and Password authenticate with HTTP basic authentication.

Basic authentication sends the password in a header that is only protected by TLS, so these belong with an https URL. Over http they are readable by anything on the path, and no warning from the server will tell you so.

21.24.2 password

User and Password authenticate with HTTP basic authentication.

Basic authentication sends the password in a header that is only protected by TLS, so these belong with an https URL. Over http they are readable by anything on the path, and no warning from the server will tell you so.

21.24.3 timeout

Timeout bounds one request.

Defaults to sixty seconds.

Longer than the FTP and SMB defaults deliberately. A WebDAV collection listing is a single PROPFIND that the server may spend a long time assembling for a large directory, and there is no partial result to fall back on.

22 Destination Reference

Every way a channel can send messages, and every key each one accepts. A channel may have any number of destinations, and they are independent: one failing does not stop the others.

22.1 AMQP Destination

AMQPDestination sends to a queue or topic, and reports success only once the broker has accepted the message.

Defined as AMQPDestination in internal/config.

Key	Type	Required	Default
content_type	text	no	the channel's data type

22.1.1 content_type

ContentType is sent with each message.

Defaults to the channel's data type.

22.2 Azure Blob Destination

AzureBlobDestination writes each message as a block blob.

Defined as AzureBlobDestination in internal/config.

Key	Type	Required	Default
blob	text	no	\${date}/\${channel}/\${control_id}
tier	text	no	—
content_type	text	no	—

22.2.1 blob

Blob names each blob, with the S3 destination's placeholders.

Defaults to \${date}/\${channel}/\${control_id}.hl7.

22.2.2 tier

Tier is Hot, Cool, Cold or Archive.

Archive is offline: reading a blob back means rehydrating it, which takes hours.

22.3 Broker Destination

BrokerDestination publishes messages to a message broker.

Mirth's JMS Writer.

Defined as BrokerDestination in internal/config.

Key	Type	Required	Default
addr	text	yes	—
destination	text	yes	—
login	text	no	—
passcode	text	no	—
host	text	no	the address's host
content_type	text	no	text/plain for HL7
headers	mapping of text to text	no	—
persistent	true or false (a block, omit to leave off)	no	true
timeout	duration, such as 30s or 5m	no	—
tls	tlsconf.Settings (a block, omit to leave off)	no	—

22.3.1 addr

Addr is the broker's host and port.

Required.

22.3.2 destination

Destination is the queue or topic to publish to.

Required.

22.3.3 login

Login and Passcode authenticate.

Most brokers refuse an anonymous connection.

22.3.4 passcode

Login and Passcode authenticate.

Most brokers refuse an anonymous connection.

22.3.5 **host**

Host is the virtual host.

Defaults to the address's host.

Worth setting explicitly on a broker that hosts several virtual hosts, where the default lands the publish on whichever one answers to the server's own name - a queue that exists, accepts the message, and is read by nobody.

22.3.6 **content_type**

ContentType is sent with each message.

Defaults to text/plain for HL7.

22.3.7 **headers**

Headers are sent with every message.

Configuration only, never message content. A header taken from a field would put a patient identifier into broker metadata that is logged, indexed and visible to every other application on the queue.

22.3.8 **persistent**

Persistent asks the broker to survive a restart.

Defaults to true.

True because a non-persistent message is lost when the broker restarts, and that is not a reasonable default for clinical data. False exists for a genuinely transient feed - a monitoring heartbeat - and has to be chosen.

22.4 **CDA Destination**

CDADestination configures a cda destination: it takes the clinical document carried inside an HL7 v2 message, converts it, and writes or posts the result.

This exists because a document almost never arrives on its own. It arrives base64-encoded inside an MDM^T02, and every engine that treats that message as an opaque blob leaves the document unread. Owning both ends of that is the point.

Defined as CDADestination in internal/config.

Key	Type	Required	Default
url	text	no	—
dir	text	no	—
write	text	no	—
version	text	no	R5
identifier_systems	mapping of text to text	no	—
require_agreement	true or false	no	—
on_no_document	text	no	—
headers	mapping of text to text	no	—
bearer_token	text	no	—

22.4.1 url

URL is a FHIR server to post the converted bundle to.

Leave it empty to write to Dir instead. One of the two is required.

22.4.2 dir

Dir writes the output to files instead of posting it.

Useful for a landing zone another process picks up, and for seeing what conversion produces before pointing it at a live server.

22.4.3 write

Write selects what is written or posted:

fhir the converted transaction bundle (default) document the original document bytes, unchanged both both, with the document alongside the bundle

"document" is worth having on its own: a site that wants its CDAs on disk for a records team does not need the FHIR conversion at all.

22.4.4 identifier_systems

IdentifierSystems maps an OID root from the document to a URI, so an MRN becomes namespaced.

A CDA identifies people by OID and FHIR by URI, and nothing can derive one from the other.

22.4.5 **require_agreement**

RequireAgreement refuses a document whose narrative and coded entries contradict each other.

Off by default, and deliberately so. The check is valuable but it is a judgement about a document somebody else authored, and a channel that silently drops real clinical documents because a sender's C-CDA generator is sloppy is worse than one that passes them through with a warning. A site that has looked at its own findings and decided it wants the gate can turn it on.

22.4.6 **on_no_document**

OnNoDocument decides what happens when the message carries no clinical document:

skip deliver nothing and report success (default) fail treat it as a delivery failure

The default is skip, because a channel carrying a mixed ADT and MDM feed would otherwise fail every admission. A channel dedicated to documents should set fail, since a document message with no document in it is a real problem at the sender.

22.4.7 **bearer_token**

BearerToken is sent as an Authorization header.

Prefer an environment variable reference over a literal in a file that goes into git.

22.5 **Channel Destination**

ChannelDestination routes to another channel in the same server.

Defined as ChannelDestination in internal/config.

Key	Type	Required	Default
name	text	yes	—

22.6 **DICOM Destination**

DICOMDestination sends imaging objects as a C-STORE service class user.

Mirth calls this the DICOM Sender.

Defined as DICOMDestination in internal/config.

Key	Type	Required	Default
address	text	yes	—
called_ae	text	no	—
calling_ae	text	no	—
transfer_syntaxes	list of text	no	—
timeout	duration, such as 30s or 5m	no	—
tls	tlsconf.Settings (a block, omit to leave off)	no	—

22.6.1 address

Address is the remote host and port.

Required.

22.6.2 called_ae

CalledAE is what the remote calls itself, and CallingAE what this end calls itself.

Both matter. Sites configure a PACS to accept one specific calling title and reject everything else, so a mismatch is the commonest reason a first connection fails - and the failure looks like a network problem.

22.6.3 calling_ae

CalledAE is what the remote calls itself, and CallingAE what this end calls itself.

Both matter. Sites configure a PACS to accept one specific calling title and reject everything else, so a mismatch is the commonest reason a first connection fails - and the failure looks like a network problem.

22.6.4 transfer_syntaxes

TransferSyntaxes restricts what to offer, in preference order.

Empty offers the object's own encoding and then implicit VR little endian as a fallback.

22.6.5 timeout

Timeout bounds one store operation.

Zero applies a default.

22.7 Database Destination

DatabaseDestination writes each message to a database.

Defined as DatabaseDestination in internal/config.

Key	Type	Required	Default
driver	text	yes	—
dsn	text	yes	—
statement	text	yes	—
params	list of text	no	—
timeout	duration, such as 30s or 5m	no	30s
max_open_conns	whole number	no	4

22.7.1 driver

Driver names the database.

See SupportedDrivers.

22.7.2 statement

Statement is executed once per message, with Params bound to it in order.

Placeholders are the driver's own: \$1 for postgres, ? for mysql and sqlite, @p1 for sqlserver. Perfuse does not rewrite them, because a translation layer that got a placeholder wrong would bind a patient's name to the wrong column and the result would look like valid data.

22.7.3 params

Params are HL7 paths or \${...} expressions, bound to the statement in order.

Values are always bound as parameters and never interpolated into the SQL. That is not only about injection: O'Brien is a common name, and a concatenated query breaks on the apostrophe.

22.7.4 timeout

Timeout bounds a single statement.

Defaults to 30s.

22.7.5 max_open_conns

MaxOpenConns caps the pool.

Defaults to 4.

22.8 Destination

Destination is one place a channel sends messages.

Defined as Destination in internal/config.

Key	Type	Required	Default
name	text	yes	—
type	DestinationType	yes	—
enabled	true or false (a block, omit to leave off)	no	—
filter	text	no	—
address	text	no	—
dir	text	no	—
tls	tlsconf.Settings (a block, omit to leave off)	no	—
fhir	FHIRDestination (a block, omit to leave off)	no	—
cda	CDADestination (a block, omit to leave off)	no	—
http	HTTPDestination (a block, omit to leave off)	no	—
database	DatabaseDestination (a block, omit to leave off)	no	—
sftp	SFTPDestination (a block, omit to leave off)	no	—
tcp	TCPDest (a block, omit to leave off)	no	—
smtp	SMTPDestination (a block, omit to leave off)	no	—
channel	ChannelDestination (a block, omit to leave off)	no	—
s3	S3Destination (a block, omit to leave off)	no	—
ftp	FTPDestination (a block, omit to leave off)	no	—
document	DocumentDestination (a block, omit to leave off)	no	—
soap	SOAPDestination (a block, omit to leave off)	no	—
dicom	DICOMDestination (a block, omit to leave off)	no	—
javascript	JavaScriptDestination (a block, omit to leave off)	no	—
broker	BrokerDestination (a block, omit to leave off)	no	—
kafka	KafkaDestination (a block, omit to leave off)	no	—

Key	Type	Required	Default
sqs	SQSDestination (a block, omit to leave off)	no	—
sns	SNSDestination (a block, omit to leave off)	no	—
amqp	AMQPDestination (a block, omit to leave off)	no	—
azure_blob	AzureBlobDestination (a block, omit to leave off)	no	—
mqtt	MQTTDestination (a block, omit to leave off)	no	—
ibm_mq	IBMMQDestination (a block, omit to leave off)	no	—
pubsub	PubSubDestination (a block, omit to leave off)	no	—
gcs	GCSDestination (a block, omit to leave off)	no	—
elr	ELRDestination (a block, omit to leave off)	no	—
response_transformer	text	no	—
timeout	duration, such as 30s or 5m	no	—
retry	Retry	no	—
queue	QueueConfig (a block, omit to leave off)	no	—

22.8.1 name

Name identifies the destination in logs and metrics.

Required within a channel.

22.8.2 elr

ELR makes this a public health lab reporting destination: each lab result (ORU^R01) is reshaped into an ELR 2.5.1 message holding only the orders with a reportable result, and that is what is delivered; a message with nothing reportable is not sent at all.

Applies to destinations that deliver an HL7 v2 message as it is (mllp, http, file, sftp and the rest), on an hl7 channel.

22.8.3 response_transformer

ResponseTransformer inspects what the receiver said back and may mark the delivery failed.

It exists because "did this arrive" is often not a question the transport can answer: an MLLP receiver returns an application acknowledgement whose meaning is in its text, and an HTTP receiver returns 200

with an error document. In both cases the transport succeeded and the message did not arrive.

Only meaningful on a destination that receives a reply. On one that cannot, it is refused at load rather than left never running.

22.8.4 queue

Queue makes delivery durable: a message that cannot be delivered now is kept on disk and retried until it can be.

Off by default.

22.9 Document Destination

DocumentDestination renders each message into a document.

Defined as DocumentDestination in internal/config.

Key	Type	Required	Default
dir	text	yes	—
format	DocumentFormat	no	pdf
template	text	yes	—
title	text	no	—
file_name	text	no	—
font_size	number	no	10
landscape	true or false	no	—
temp_suffix	text	no	"
timeout	duration, such as 30s or 5m	no	30s

22.9.1 format

Format is pdf or text.

Defaults to pdf.

22.9.2 template

Template is the document body, with \${PID-5.1} style references to message paths.

Required. There is no sensible default: a document with no template would either be the raw message, which nobody wants printed, or empty.

22.9.3 font_size

FontSize in points.

Defaults to 10, which fits 80 characters across A4.

22.9.4 temp_suffix

TempSuffix is appended while writing and removed by a rename.

Defaults to ".part".

The same courtesy as every other file destination, and it matters more here: a print watcher picking up a half-written PDF produces a page of nothing, and nobody investigates a blank page.

22.9.5 timeout

Timeout bounds one write.

Defaults to 30s.

22.10 ECR Destination

ECRDestination configures electronic case reporting.

Defined as ECRDestination in internal/config.

Key	Type	Required	Default
rctc	text	no	—
source	text	yes	—
facility	ECRFacility	yes	—

22.10.1 rctc

RCTC is a FHIR ValueSet, or a Bundle of them such as the eRSD, holding the trigger codes.

Without it Perfuse's short built-in sample is used, which is for testing: agencies expect the RCTC.

22.11 ELR Destination

ELRDestination configures electronic laboratory reporting.

Defined as ELRDestination in internal/config.

Key	Type	Required	Default
config	text	yes	—
rctc	text	no	—

22.11.1 config

Config is the ELR file `perfuse elr -config` reads: the sending and receiving applications and facilities by OID, the processing mode, and what fills the gaps a lab feed leaves (ordering facility, performing lab, assigning authorities).

A relative path is read from the channel file's directory.

22.11.2 rctc

RCTC is a FHIR ValueSet, or a Bundle of them such as the eRSD, holding the trigger codes.

Without it Perfuse's short built-in sample is used, which is for testing: agencies expect the RCTC. Relative to the channel file, as Config.

22.12 FHIR Destination

FHIRDestination configures conversion to FHIR and delivery to a FHIR server.

Defined as FHIRDestination in `internal/config`.

Key	Type	Required	Default
url	text	yes	—
version	text	no	R5
timezone	text	no	—
default_identifier_system	text	no	—
identifier_systems	mapping of text to text	no	—
claim_us_core	true or false	no	—
validate_before_send	true or false (a block, omit to leave off)	no	true: sending a resource a server will reject wastes a retry budget and hides the real problem behind a transport error
reject_on_warning	true or false	no	—
headers	mapping of text to text	no	—
bearer_token	text	no	—
auth	FHIRAuth (a block, omit to leave off)	no	—
ecr	ECRDestination (a block, omit to leave off)	no	—

22.12.1 url

URL is the base URL of the FHIR server, for example "https://fhir.example.org/fhir".

The bundle is posted to it directly.

22.12.2 version

Version is the FHIR release to produce: R4, R4B or R5. Defaults to R5, the latest published release.

A deployment talking to an EHR almost always wants R4 and should say so, because silently downgrading would hide a real interoperability decision.

22.12.3 **timezone**

Timezone is applied to HL7 v2 timestamps that carry no offset. v2 permits a bare local time and FHIR does not, so something has to supply one.

When unset, the offset the sender put on MSH-7 is used, since v2 presumes a bare time is the sender's local time; a message whose MSH-7 has no offset either falls back to UTC.

22.12.4 **default_identifier_system**

DefaultIdentifierSystem namespaces identifiers whose assigning authority has no configured URI.

Without a system, an MRN is ambiguous between facilities.

22.12.5 **claim_us_core**

ClaimUSCore adds the US Core 9.0.0 profile to each resource produced that has been checked against it and conforms.

A resource that falls short is sent without the claim, with a note saying why.

22.12.6 **validate_before_send**

ValidateBeforeSend refuses to post a bundle that fails validation.

Defaults to true: sending a resource a server will reject wastes a retry budget and hides the real problem behind a transport error.

22.12.7 **reject_on_warning**

RejectOnWarning also refuses on warnings, such as a missing US Core identifier.

Off by default, because a warning is a judgement about profile conformance rather than about validity.

22.12.8 **bearer_token**

BearerToken is sent as an Authorization header.

Prefer an environment variable reference, \${NAME}, over a literal in a file that goes into git.

22.12.9 **ecr**

ECR makes this a public health case reporting destination.

Each message is checked for trigger codes; one that has any is sent as an eCR 2.1 eICR, wrapped in an eCR message, to url's \$process-message, and one with none is not sent at all.

22.13 **FTP Destination**

FTPDestination uploads a file per message.

Defined as FTPDestination in internal/config.

Key	Type	Required	Default
host	text	yes	21
user	text	no	—
password	text	no	—
security	FTPSecurity	no	explicit rather than none
insecure_skip_verify	true or false	no	—
allow_clear_password	true or false	no	—
dir	text	yes	—
file_name	text	no	—
temp_suffix	text	no	"
framed	true or false	no	—
timeout	duration, such as 30s or 5m	no	60s

22.13.1 host

Host is the server, with an optional port.

Defaults to 21, or 990 for implicit TLS.

22.13.2 user

User and Password authenticate.

An empty user means anonymous.

22.13.3 password

User and Password authenticate.

An empty user means anonymous.

22.13.4 security

Security selects TLS: explicit, implicit or none.

Defaults to explicit rather than none, because defaulting to plain FTP would make the insecure choice the quiet one, and a site that genuinely has no TLS should have to write that down.

22.13.5 **insecure_skip_verify**

InsecureSkipVerify accepts any certificate.

Needed more often than anybody would like: these servers frequently have a self-signed certificate generated years ago that nobody can reissue. Named separately so choosing it is deliberate and visible in the file rather than hidden inside a "security: relaxed" mode.

22.13.6 **allow_clear_password**

AllowClearPassword permits a password over plain FTP.

Refused by default, because that sends a credential across the network in clear text on every single delivery. Some analysers genuinely have no TLS, so it has to be possible - but it has to be written down, so that it is a decision somebody made rather than a default they inherited. The name is deliberately uncomfortable to type.

22.13.7 **temp_suffix**

TempSuffix is appended while the file is being written and removed by a rename once it is complete.

Defaults to ".part".

Whoever collects these files is usually a scheduled job that takes whatever it finds, and without the rename it eventually takes half a message.

22.13.8 **timeout**

Timeout bounds one transfer.

Defaults to 60s.

22.14 **GCS Destination**

GCSDestination writes each message as an object.

Defined as GCSDestination in internal/config.

Key	Type	Required	Default
object	text	no	\${date}/\${channel}/\${control_id}
storage_class	text	no	—
content_type	text	no	—

22.14.1 **object**

Object is the object name, with the same placeholders as an s3 key.

Defaults to \${date}/\${channel}/\${control_id}.hl7.

22.15 HTTP Destination

HTTPDestination posts messages to an HTTP endpoint.

The commonest thing a Mirth channel does that Perfuse could not, and the commonest blocker the translator reports. Most uses are unglamorous: post the message to an internal API, post it to a web service that wraps a legacy system, post it to something somebody wrote in an afternoon.

Defined as HTTPDestination in internal/config.

Key	Type	Required	Default
url	text	yes	—
method	text	no	—
content_type	text	no	—
headers	mapping of text to text	no	—
bearer_token	text	no	—
username	text	no	—
password	text	no	—
tls	tlsconf.Settings (a block, omit to leave off)	no	—
success_status	list of whole number	no	any 2xx
fail_on_body	list of text	no	—
follow_redirects	true or false	no	—

22.15.1 url

URL is where to post.

Required.

22.15.2 method

Method defaults to POST.

PUT is allowed for an endpoint that wants it.

GET and DELETE are refused: this sends a message body, and a transport that silently dropped the message because of a method choice would be a very quiet way to lose data.

22.15.3 **content_type**

ContentType defaults to application/hl7-v2+er7, which is the registered type for a pipe-delimited HL7 v2 message.

Many endpoints want text/plain instead.

22.15.4 **bearer_token**

BearerToken is sent as an Authorization header.

Prefer an environment variable reference over a literal in a file that goes into git.

22.15.5 **success_status**

SuccessStatus lists the status codes that count as delivered.

Defaults to any 2xx.

Configurable because endpoints disagree about what success looks like: some answer 200 with an error in the body, and some answer 202 for a queue they have accepted the message into.

22.15.6 **fail_on_body**

FailOnBody treats a response containing any of these strings as a failure, whatever the status code.

This exists because an endpoint answering 200 with "ERROR: patient not found" in the body is common, and treating it as delivered means the message is gone and nobody knows.

22.15.7 **follow_redirects**

FollowRedirects allows the request to be redirected.

Off by default: a redirect on a write would repost clinical data somewhere the configuration never named.

22.16 **IBMMQ Destination**

IBMMQDestination puts each message on a queue.

Defined as IBMMQDestination in internal/config.

Key	Type	Required	Default
non_persistent	true or false	no	—

22.16.1 **non_persistent**

NonPersistent sends messages that do not survive a queue manager restart.

Off by default.

22.17 Java Script Destination

JavaScriptDestination runs a script instead of sending anywhere.

Mirth's JavaScript Writer, and it is used far more than its position in a feature list suggests. It is the escape hatch: when a site needs to talk to something no connector covers, or apply logic no transformation expresses, they write a JavaScript Writer. A migration that cannot run those cannot move the channels that have them, which in a mature Mirth installation is a large fraction.

Deliberately not a general scripting host. The script gets the message and the same helpers a transformer gets; what it does not get is a way to reach the filesystem or the network, because that is what the other destinations are for and a script that opens its own socket is invisible to every retry, queue and metric in the engine.

Defined as JavaScriptDestination in `internal/config`.

Key	Type	Required	Default
script	text	yes	—
timeout	duration, such as 30s or 5m	no	—
success_on_undefined	true or false (a block, omit to leave off)	no	true

22.17.1 script

Script is the code to run.

Required.

22.17.2 timeout

Timeout bounds one execution.

Zero applies a default.

Bounded because a destination script is on the delivery path: an infinite loop in one does not merely fail a message, it holds the queue behind it.

22.17.3 success_on_undefined

SuccessOnUndefined treats a script that returns nothing as success.

Defaults to true.

True is the right default and worth explaining. Mirth's JavaScript Writers overwhelmingly do their work and return nothing, so requiring an explicit return would break every one of them on import. Setting it false is for a site that wants a script to be explicit about whether it worked.

22.18 Kafka Destination

KafkaDestination publishes messages to a Kafka topic.

Defined as KafkaDestination in internal/config.

Key	Type	Required	Default
brokers	list of text	yes	—
topic	text	yes	—
key	text	no	—
acks	text	no	all
compression	text	no	snappy
headers	mapping of text to text	no	—
timeout	duration, such as 30s or 5m	no	thirty seconds
sasl	KafkaSASL (a block, omit to leave off)	no	—
tls	tlsconf.Settings (a block, omit to leave off)	no	—

22.18.1 brokers

Brokers is the bootstrap server list.

Required.

22.18.2 topic

Topic is where to publish.

Required.

22.18.3 key

Key is a template for the record key, which decides the partition.

This is the field that makes Kafka safe for a clinical feed, and it is worth being precise about why. Kafka guarantees order within a partition and nowhere else. Records with the same key always land in the same partition, so keying on the patient identifier keeps that patient's events in order while letting different patients go in parallel.

Without a key there is no such guarantee. Measured against a real broker, thirty unkeyed records went to one partition rather than spreading: the client keeps a batch together and picks a new partition between batches. So the ordering holds by accident and breaks at a boundary nobody can see, which is

worse than breaking consistently - an A03 discharge read before its A01 admission, intermittently, under load.

An HL7 path, evaluated against the outgoing message: PID-3.1 keys by patient.

22.18.4 **acks**

Acks is how many brokers must confirm a write: "all", "leader" or "none".

Defaults to all, which is the only setting that survives a broker failing between the write and the replication. "none" is fire-and-forget and will lose messages; it exists because somebody moving non-clinical telemetry may legitimately want it, and refusing it outright would mean they wrote their own producer instead.

22.18.5 **compression**

Compression is "none", "gzip", "snappy", "lz4" or "zstd".

Defaults to snappy.

On by default because HL7 is highly compressible text and the wire is usually the constraint. Snappy rather than zstd as the default: it is the cheapest in CPU, and a connector on the delivery path should not be the thing that saturates a core.

22.18.6 **headers**

Headers are record headers to set.

Values are templates evaluated per message.

22.18.7 **timeout**

Timeout bounds one publish.

Defaults to thirty seconds.

22.19 **MQTT Destination**

MQTTDestination publishes to a topic.

Defined as MQTTDestination in internal/config.

Key	Type	Required	Default
topic	text	yes	—
retain	true or false	no	—

22.19.1 **topic**

Topic is published to.

It may not hold wildcards.

22.20 Pub Sub Destination

PubSubDestination publishes to a Pub/Sub topic.

Defined as `PubSubDestination` in `internal/config`.

Key	Type	Required	Default
topic	text	yes	—
order_by	text	no	—

22.21 S3 Destination

S3Destination writes each message as an object.

Defined as `S3Destination` in `internal/config`.

Key	Type	Required	Default
bucket	text	yes	—
region	text	yes	—
key	text	no	a date-partitioned path plus the control id
access_key_id	text	no	—
secret_access_key	text	no	—
session_token	text	no	—
endpoint	text	no	—
path_style	true or false	no	—
server_side_encryption	text	no	—
content_type	text	no	application/hl7-v2 for HL7 and text/plain otherwise
framed	true or false	no	—
timeout	duration, such as 30s or 5m	no	60s
storage_class	text	no	—
format	text	no	—

22.21.1 key

Key templates the object key, using the same `${date}`, `${timestamp}`, `${control_id}`, `${message_type}` and `${channel}` placeholders as the file and SFTP destinations.

Defaults to a date-partitioned path plus the control id.

Date partitioning by default because the alternative - every object in one flat prefix - makes a bucket that is slow to list and impossible to lifecycle by age, and nobody discovers that until there are four million objects in it.

22.21.2 access_key_id

AccessKeyID and SecretAccessKey authenticate.

Either may name an environment variable instead of holding the value, using the form `${NAME}`.

22.21.3 **secret_access_key**

AccessKeyID and SecretAccessKey authenticate.

Either may name an environment variable instead of holding the value, using the form \${NAME}.

22.21.4 **path_style**

PathStyle addresses the bucket as a path rather than a subdomain.

Most S3-compatible stores need this.

22.21.5 **content_type**

ContentType labels the object.

Defaults to application/hl7-v2 for HL7 and text/plain otherwise.

22.21.6 **timeout**

Timeout bounds one upload.

Defaults to 60s.

22.21.7 **storage_class**

StorageClass is the S3 storage class each object is written with: STANDARD_IA, INTELLIGENT_TIERING, GLACIER_IR, GLACIER, DEEP_ARCHIVE and so on.

This is how an archive goes straight to Glacier - S3 has no separate Glacier API to call.

22.21.8 **format**

Format is hl7 (the default: each object is the message) or ndjson: each object is one JSON line with the message and the fields worth querying, under date-partitioned keys, so Amazon Athena can query the archive.

See perfuse athena.

22.22 **SFTP Destination**

SFTPDestination writes each message to a file on an SFTP server.

Defined as SFTPDestination in internal/config.

Key	Type	Required	Default
host	text	yes	22
user	text	yes	—
password	text	no	—
key_file	text	no	—
key_passphrase	text	no	—
known_hosts_file	text	no	—
insecure_skip_host_key_check	true or false	no	—
dir	text	yes	—
file_name	text	no	a timestamp plus the control ID
temp_suffix	text	no	"
framed	true or false	no	—
append	true or false	no	—
timeout	duration, such as 30s or 5m	no	60s

22.22.1 host

Host is the server, with an optional port.

Defaults to 22.

22.22.2 password

Password authenticates with a password.

Prefer KeyFile.

22.22.3 file_name

FileName templates the name.

Defaults to a timestamp plus the control ID.

22.22.4 **temp_suffix**

TempSuffix is appended while a file is being written, and removed by a rename once it is complete.

Defaults to ".part".

The same courtesy this connector needs on the way in. Whoever collects these files has the identical problem, and a rename within a directory is atomic on every server worth using.

22.22.5 **timeout**

Timeout bounds a single connection.

Defaults to 60s.

22.23 **SMTP Destination**

SMTPDestination configures delivery by email.

Defined as SMTPDestination in internal/config.

Key	Type	Required	Default
host	text	yes	—
from	text	yes	—
to	list of text	yes	—
cc	list of text	no	—
bcc	list of text	no	—
subject	text	no	—
body	text	no	—
attach	true or false	no	—
attach_name	text	no	the control ID with a
username	text	no	—
password	text	no	—
starttls	true or false (a block, omit to leave off)	no	—
insecure_skip_verify	true or false	no	—
timeout	duration, such as 30s or 5m	no	—

22.23.1 host

Host is the mail server, with an optional port.

Port 587 is assumed, which is submission with STARTTLS - the port a modern server expects a client on. Port 25 is server-to-server and is usually either blocked or unauthenticated.

22.23.2 from

From is the envelope sender.

Required, because a message with no sender is discarded silently by a great deal of mail infrastructure, which makes it the hardest possible failure to diagnose.

22.23.3 to

To, CC and BCC are recipients.

At least one To is required.

22.23.4 **cc**

To, CC and BCC are recipients.

At least one To is required.

22.23.5 **bcc**

To, CC and BCC are recipients.

At least one To is required.

22.23.6 **body**

Body is the message text.

Field references are substituted, as in Subject. When it is empty the message itself is the body.

22.23.7 **attach_name**

AttachName names the attachment.

Defaults to the control ID with a .hl7 suffix.

22.23.8 **username**

Username and Password authenticate to the server.

Both or neither.

22.23.9 **password**

Username and Password authenticate to the server.

Both or neither.

22.23.10 **starttls**

StartTLS upgrades the connection before authenticating.

It defaults to on, and turning it off with a username set is refused rather than allowed: sending a password over an unencrypted connection is not a trade-off worth offering.

22.23.11 **insecure_skip_verify**

InsecureSkipVerify accepts any certificate.

Named unambiguously because that is what it does.

22.24 **SNS Destination**

SNSDestination publishes each message to an SNS topic.

Defined as SNSDestination in internal/config.

Key	Type	Required	Default
topic_arn	text	yes	—
subject	text	no	—
group_by	text	no	—

22.24.1 subject

Subject is sent with email subscriptions.

Optional; `${message_type}` and `${control_id}` are filled in.

22.25 SOAP Destination

SOAPDestination posts a SOAP request per message.

Defined as SOAPDestination in `internal/config`.

Key	Type	Required	Default
url	text	yes	—
action	text	no	—
version	text	no	1
body	text	yes	—
header	text	no	—
headers	mapping of text to text	no	—
username	text	no	—
password	text	no	—
tls	tlsconf.Settings (a block, omit to leave off)	no	—
fault_is_success	list of text	no	—
timeout	duration, such as 30s or 5m	no	30s

22.25.1 action

Action is the SOAPAction.

Required by most 1.1 services.

22.25.2 **body**

Body is the XML that goes inside the envelope, with \${PID-5.1} style references to message paths.

Required. There is no useful default: the element names come from the service's own schema and no engine can guess them.

22.25.3 **header**

Header is XML for the envelope's Header element.

Omitted entirely when empty, because some services reject an empty one.

22.25.4 **fault_is_success**

FaultIsSuccess lists fault codes that count as delivered.

Needed more often than it should be. A service that answers a resend with "already submitted" is telling you the message arrived, and without this a site has to write a response transformer to say so - or worse, watches a queue retry forever against a receiver that already has the message.

22.25.5 **timeout**

Timeout bounds one call.

Defaults to 30s.

22.26 **SQS Destination**

SQSDestination sends each message to an SQS queue.

Defined as SQSDestination in internal/config.

Key	Type	Required	Default
queue_url	text	yes	—
group_by	text	no	—

22.26.1 **queue_url**

QueueURL names the queue.

One ending .fifo is a FIFO queue, and is grouped and deduplicated.

22.27 **TCP Dest**

TCPDest sends messages to a socket with configurable framing.

Defined as TCPDest in internal/config.

Key	Type	Required	Default
address	text	yes	—
max_message_size	whole number	no	—
timeout	duration, such as 30s or 5m	no	—
expect_reply	true or false	no	—
keep_alive	true or false	no	—
tls	tlsconf.Settings (a block, omit to leave off)	no	—

22.27.1 expect_reply

ExpectReply waits for bytes back before reporting success.

Off by default, and worth thinking about. With it off, "delivered" means the bytes reached the operating system's send buffer, which a peer that crashed a moment later never read. With it on, a peer that never replies makes every message time out.

22.27.2 keep_alive

KeepAlive holds one connection open across messages rather than dialling per message.

Most device endpoints expect this; some refuse a second message on the same connection. Off by default because a connection per message is the behaviour that works everywhere, at the cost of a handshake each time.

23 Format Reference

Options governing how messages are parsed and written, for the formats that need more than a name.

23.1 Attachments

Attachments configures moving large payloads out of a message.

A message carrying a scanned report or a PDF is ordinary in document workflows, and the payload is routinely hundreds of times larger than the message around it. Left inline it means the store grows at the rate of the documents rather than the traffic, a message queued for five destinations is copied five times, and the interface streams megabytes to show somebody a patient name.

Mirth calls these attachment handlers. This is the equivalent, and it is opt-in per channel because a feed of plain ADT messages has nothing to extract and should not pay for the machinery.

Defined as Attachments in `internal/config`.

Key	Type	Required	Default
extract	list of attach.Rule	yes	—
reassemble	true or false (a block, omit to leave off)	no	true

23.1.1 reassemble

Reassemble puts payloads back before delivery.

Defaults to true.

The default is on because a receiver expecting a document must get a document; a message containing a literal token would be filed as a report and nobody would find out until somebody opened the record.

Turning it off is for the case where the point of the channel is to strip documents - forwarding metadata to a system that has no use for the image, where sending it would be a privacy question rather than a bandwidth one.

23.2 Delimited

Delimited configures a delimited channel.

Mirth's Delimited data type: CSV, tab-separated, and the long tail of formats a laboratory analyser or a bureau service emits.

Defined as Delimited in `internal/config`.

Key	Type	Required	Default
delimiter	text	no	—
quote	text	no	a double quote
comment	text	no	—
has_header	true or false	no	—
columns	list of text	no	—
trim_space	true or false	no	—
relaxed	true or false	no	—
split	true or false (a block, omit to leave off)	no	true
keep_blank_lines	true or false	no	—
filter	text	no	—
transformations	list of steps.Step	no	—

23.2.1 delimiter

Delimiter separates columns, as a character or a name.

A name is accepted because a tab cannot be written into a YAML file unambiguously, and a configuration file containing an invisible character is unreadable to whoever opens it next.

23.2.2 quote

Quote wraps values containing the delimiter.

Defaults to a double quote; "none" disables it.

23.2.3 split

Split makes each row its own message.

Defaults to true.

This is the decision that shapes a delimited channel and it deserves saying out loud. A file of five thousand results is either one message or five thousand. One message means one acknowledgement, one entry in the browser, and one failure that takes the whole file with it. Five thousand means each row is filtered, transformed, delivered and retried on its own, and a single bad row does not stop the rest.

Splitting is the default because per-row handling is what almost every site wants and because the alternative is discovered late: a channel that looked fine in testing with a three-row file fails an entire

night's transfer over one malformed line.

23.2.4 **keep_blank_lines**

KeepBlankLines stops blank lines being skipped.

Skipping is the default, because every file ends in a newline and a record of empty fields is not a record. This exists for the rare format where a blank line is meaningful.

23.2.5 **filter**

Filter decides which messages continue, written against column names.

Here rather than on the channel because the channel-level filter compiles HL7 paths. Two settings named filter would be confusing, but one setting that means different things depending on dataType is worse: a delimited channel that silently compiled its filter as HL7 would match nothing and drop everything, which is the failure the top-level refusal exists to prevent.

23.2.6 **transformations**

Transformations are the declarative steps, addressing columns rather than HL7 fields.

Separate from the channel's transformations for the same reason as Filter, and it is the arrangement X12 and NCPDP already use.

23.3 **HL7v3 Options**

HL7v3Options are the settings that only apply to an HL7 v3 channel.

Defined as HL7v3Options in internal/config.

Key	Type	Required	Default
filter	text	no	—
acknowledge	true or false (a block, omit to leave off)	no	—
sender_device	text	no	—
sender_oid	text	no	—
transformations	list of hl7v3.Step	no	—

23.3.1 **filter**

Filter excludes messages, using v3 paths.

Its own field rather than the channel's ordinary filter, because the two are different languages against different message models. Sharing the field would mean a v2 filter silently never matching on a v3 channel, or a v3 filter failing to parse as v2 - and the second is only better because it fails loudly.

A channel that sets both is refused at load, since one of them would have to be ignored and there is no honest way to choose.

23.3.2 **acknowledge**

Acknowledge sends an MCCI_IN000002UV01 acknowledgement back.

On by default, unlike X12. A v3 interaction over a synchronous transport expects an acknowledgement - the sending application is generally waiting on one - and a sender that receives nothing will usually retry, which is how a patient gets registered three times.

A pointer so that "not set" and "set to false" are different: the default is on, and somebody turning it off should have said so.

23.3.3 **sender_device**

SenderDevice is the device identifier we put in an acknowledgement.

Required when acknowledging. A v3 acknowledgement names the device it comes from, and a receiver that does not recognise ours may discard it - which looks exactly like not sending one.

23.3.4 **transformations**

Transformations change the content of a v3 document.

Its own field rather than the channel's ordinary transformations, for the same reason the filter is: they are different vocabularies against different message models. The v2 steps address segments and fields, which a v3 document does not have, so sharing the field would mean a step that silently did nothing.

The step names match the v2 ones wherever the meaning matches, so somebody who has written one channel can write the other. What is different is what v3 needs and v2 cannot express - chiefly that removing a value has three distinct meanings, so clear, nullflavor and remove are three steps rather than one.

23.4 **TCP Framing**

TCPFraming is how message boundaries are found on a raw socket.

Shared by the TCP source and the TCP destination, and by the serial source, because the framings are the same wherever the bytes come from. A laboratory analyser that speaks STX/ETX over a socket speaks STX/ETX over a serial cable too.

Defined as TCPFraming in `internal/config`.

Key	Type	Required	Default
framing	text	yes	—
delimiter	text	no	—
start_block	text	no	—
keep_delimiter	true or false	no	—
record_length	whole number	no	—
trim_padding	true or false (a block, omit to leave off)	no	—
length_bytes	whole number	no	—
big_endian	true or false	no	—
length_includes_header	true or false	no	—

23.4.1 framing

Framing is mllp, delimited, fixed, length or whole.

Required, with no default. Reading a stream with the wrong framing produces messages that look plausible rather than an error, so guessing is worse than asking.

23.4.2 start_block

StartBlock optionally begins a message.

Bytes before it are discarded as noise from a partial connection.

23.4.3 keep_delimiter

KeepDelimiter includes the delimiter in the message.

Off by default: it is framing, not content.

23.4.4 trim_padding

TrimPadding removes trailing spaces and NULs from a fixed record.

On by default.

23.5 X12 Options

X12Options are the settings that only apply to an X12 channel.

Defined as X12Options in internal/config.

Key	Type	Required	Default
envelope	EnvelopePolicy	no	require
split	true or false	no	—
acknowledge	text	no	—
ack_sender_id	text	no	—
ack_sender_qualifier	text	no	—
transformations	list of x12.Step	no	—

23.5.1 envelope

Envelope says what to do when the self-declared counts do not match what is present.

Defaults to require.

23.5.2 split

Split sends one message per transaction set instead of one per file.

Off by default. A single 837 can carry hundreds of claims, so turning this on multiplies everything downstream - message counts, acknowledgements, rows, alert thresholds - and that should be a decision somebody made rather than a default they inherited.

23.5.3 acknowledge

Acknowledge says which acknowledgement to send back: 999, 997, ta1, or none.

Which one is a property of the trading partner relationship rather than of the message, so it has to be configured. A 999 supersedes a 997 for HIPAA transactions, but plenty of partners - older payer connections especially - are set up to expect a 997 and will treat a 999 as an unrecognised file. Sending the wrong one is worse than sending none, because it answers a question nobody asked and leaves the real one open.

Empty means none, which is the safe default: a partner who is not expecting an acknowledgement and receives one may treat it as an unsolicited interchange.

23.5.4 ack_sender_id

AckSenderId and AckSenderQualifier are our own interchange identifier, becoming ISA06 and ISA05 of any acknowledgement.

Required whenever Acknowledge is set, and refused at load otherwise. These must be the values the trading partner has configured for us and there is no way to guess them: an interchange whose ISA06 the partner does not recognise is discarded before anybody reads it, so a wrong value produces silence that looks exactly like not sending anything - and somebody spends a week looking in the wrong place.

23.5.5 **ack_sender_qualifier**

AckSenderID and AckSenderQualifier are our own interchange identifier, becoming ISA06 and ISA05 of any acknowledgement.

Required whenever Acknowledge is set, and refused at load otherwise. These must be the values the trading partner has configured for us and there is no way to guess them: an interchange whose ISA06 the partner does not recognise is discarded before anybody reads it, so a wrong value produces silence that looks exactly like not sending anything - and somebody spends a week looking in the wrong place.

23.5.6 **transformations**

Transformations are declarative changes applied to accepted interchanges.

Separate from the channel-level transformations for the same reason the v3 ones are: those address HL7 fields through a parser that reads segment-field-component-subcomponent, and X12 has no subcomponent, treats repeated segments as ordinary rather than exceptional, and is written CLM01 by the people who configure it. A shared parser would have to accept both notations and would then be ambiguous in both.

The step names are the same ones the HL7 and v3 steps use, so what an author has already learned carries over.

24 Behaviour Reference

Acknowledgement, queueing, retry, polling, scripting and shadow settings. These apply regardless of which transport a channel uses.

24.1 AMQP Connection

AMQPConnection is where and as whom to connect.

Defined as AMQPConnection in `internal/config`.

Key	Type	Required	Default
addr	text	yes	—
address	text	yes	—
username	text	no	—
password	text	no	—
hostname	text	no	the host of addr
tls	tlsconf.Settings (a block, omit to leave off)	no	—
timeout	duration, such as 30s or 5m	no	30s

24.1.1 username

Username and Password authenticate with SASL PLAIN; both empty means ANONYMOUS.

For Service Bus, the policy name and key.

24.1.2 password

Username and Password authenticate with SASL PLAIN; both empty means ANONYMOUS.

For Service Bus, the policy name and key.

24.1.3 hostname

Hostname is sent to the broker, which Service Bus routes on.

Defaults to the host of addr.

24.1.4 tls

TLS encrypts the connection.

Required for Service Bus and Event Hubs.

24.1.5 timeout

Timeout bounds connecting and each wait for the broker.

Defaults to 30s.

24.2 AWS Access

AWSAccess is how a connector reaches AWS: the region it signs for, the keys, and an optional endpoint override.

Credentials may be literal or `${ENV_VAR}` references, which is how they should be supplied. There is no instance-metadata credential chain, deliberately: see `internal/awsv4`.

Defined as `AWSAccess` in `internal/config`.

Key	Type	Required	Default
region	text	yes	—
access_key_id	text	no	—
secret_access_key	text	no	—
session_token	text	no	—
endpoint	text	no	—

24.2.1 secret_access_key

SecretAccessKey is the secret, literal or `${AWS_SECRET_ACCESS_KEY}`.

Prefer the reference: this file goes into version control.

24.2.2 session_token

SessionToken is for temporary credentials only.

They expire, and the connector stops working when they do.

24.3 Ack

Ack configures acknowledgement generation.

Defined as `Ack` in `internal/config`.

Key	Type	Required	Default
when	AckWhen	no	—
application	text	no	—
facility	text	no	—
include_trigger_event	true or false	no	—

24.3.1 include_trigger_event

IncludeTriggerEvent sends ACK^A01 rather than ACK.

Some receivers require it and others reject it, so it is explicit.

24.4 Azure Blob Access

AzureBlobAccess is the account, container and credential.

Defined as AzureBlobAccess in internal/config.

Key	Type	Required	Default
account	text	yes	—
container	text	yes	—
key	text	no	—
sas	text	no	—
endpoint	text	no	—

24.4.1 key

Key is the account key, or \${AZURE_STORAGE_KEY}.

Set it or sas.

24.4.2 sas

SAS is a shared access signature query string, or a \${...} reference.

Preferred: it can be limited to this container.

24.5 Contract Ref

ContractRef points a channel at its contract.

Defined as ContractRef in `internal/config`.

Key	Type	Required	Default
file	text	yes	—
check_every	text	no	15 minutes
over	whole number	no	500

24.5.1 check_every

CheckEvery is how often to re-check, as a duration.

Defaults to 15 minutes.

Effectively floored at the alert evaluation interval, which is 30 seconds by default, because the check runs from that loop rather than a timer of its own - so a contract asking for 3s gets 30s. That is deliberate: one timer is easier to reason about than two, and it keeps a contract result in step with the reading it is attached to. Worth knowing before somebody sets a short interval and concludes it is broken.

Not per message. Building a profile means reading recent traffic, and doing that on every message would make a channel's throughput depend on how much history it has - which is the kind of performance characteristic that only shows up in production, six months in.

24.5.2 over

Over is how many recent messages to profile.

Defaults to 500.

A count rather than a time window, because a quiet feed and a busy one need the same amount of evidence to justify a rate, and a time window gives them wildly different amounts.

24.6 DICOM Options

DICOMOptions configures an imaging channel.

Why the steps are named rather than paths

Every other format gets a path writer. DICOM deliberately does not, and the reason is the pixel data: an object is binary, a general tag writer can set any tag to any bytes, and a mistake there does not produce

a rejected message - it produces an image that opens and is wrong. A radiologist reading a study has no way to tell.

So the four things sites actually need are named actions with bounded effects: de-identify, rewrite an AE title, strip private tags, set the institution. Each knows which tags it touches, and none can reach the pixel data.

Why this block exists now

internal/dicom has carried steps.go and transform.go for a while - the four actions, dicom.Apply, and tests. Nothing referenced them. Five hundred lines with no caller, which the queue's own section 7 names: an implementation with no caller is indistinguishable from a feature that does not exist. This block and the engine call are the wiring, not the feature.

Defined as DICOMOptions in internal/config.

Key	Type	Required	Default
transformations	list of dicom.Step	no	—

24.7 ECR Facility

ECRFacility is the organisation sending case reports.

Defined as ECRFacility in internal/config.

Key	Type	Required	Default
name	text	yes	—
npi	text	no	—
phone	text	yes	—
line	text	no	—
city	text	yes	—
state	text	yes	—
postal_code	text	no	—

24.8 FHIR Auth

FHIRAuth is how a fhir destination authenticates to a hosted FHIR service.

Three presets, because the hosted services are where a bearer token pasted into a file stops working within the hour:

- **aws:** AWS HealthLake, which takes requests signed with Signature Version 4 for the service "healthlake" - no token at all.
- **azure:** Azure Health Data Services, which takes a Microsoft Entra ID token for the FHIR service's own audience, got with the client-credentials grant of an app registration that holds the FHIR Data Contributor role.
- **client_credentials:** any OAuth 2.0 token endpoint
- **Keycloak, Okta,** a SMART backend service with a client secret.

Tokens are fetched when needed and reused until a minute before they expire.

Defined as FHIRAuth in `internal/config`.

Key	Type	Required	Default
type	text	yes	—
region	text	no	—
access_key_id	text	no	—
secret_access_key	text	no	—
session_token	text	no	—
tenant_id	text	no	—
token_url	text	no	—
client_id	text	no	—
client_secret	text	no	—
scope	text	no	—

24.8.1 region

Region is the AWS region.

Read from a HealthLake URL when it is not given.

24.8.2 token_url

TokenURL is the OAuth token endpoint, for client_credentials.

For azure it is derived from the tenant.

24.8.3 client_id

ClientID and ClientSecret identify the application.

The secret may be a `${ENV}` reference, which is how it should be given.

24.8.4 **client_secret**

ClientID and ClientSecret identify the application.

The secret may be a `${ENV}` reference, which is how it should be given.

24.8.5 **scope**

Scope is requested with the token.

For azure it defaults to the FHIR service URL followed by `/.default`, which is its audience.

24.9 **File Poll**

FilePoll is the part of a file-collecting source that has nothing to do with the transport.

Shared by the local, FTP, SMB and WebDAV sources, and by SFTP. Every one of these settings exists because of a way files go wrong rather than a way transports differ, which is why they belong together: a site that has worked out the right `stable_for` for its analyser should not have to work it out again when the analyser moves to a share.

Defined as `FilePoll` in `internal/config`.

Key	Type	Required	Default
dir	text	yes	—
pattern	text	no	—
poll_interval	duration, such as 30s or 5m	no	—
after_read	text	no	—
move_to	text	no	—
error_dir	text	no	—
stable_for	duration, such as 30s or 5m	no	—
max_file_size	whole number	no	—
framed	true or false	no	—
raw	true or false	no	—
batch_size	whole number	no	—
sort_by	text	no	name

24.9.1 pattern

Pattern selects files by glob.

Empty means every file.

24.9.2 error_dir

ErrorDir is where a file that could not be processed goes.

Separate from MoveTo deliberately. A directory holding nothing but failures is one somebody can watch, and mixing failures into the archive means the only way to find them is to read every file.

24.9.3 raw

Raw treats the whole file as one message regardless of content.

The setting that makes this connector useful for anything other than HL7. Without it a CSV batch, an X12 claim file or a PDF is split on lines beginning MSH and produces nothing, and the error says the file is not HL7 - true but not the point, since the file was never meant to be.

24.9.4 batch_size

BatchSize bounds how many files one poll will read.

Zero means no limit.

Exists for the first poll after an outage. A directory holding forty thousand files that accumulated overnight will otherwise be read in one pass, which holds the poll loop for as long as it takes and makes the channel look hung. Reading a bounded number per poll keeps the channel responsive and the queue draining visibly.

24.9.5 **sort_by**

SortBy orders the files within a poll: name, modified or none.

Defaults to name. Ordering matters more than it looks: an ADT stream where A08 updates arrive before the A01 admission produces patients that do not exist yet. Directory listing order is arbitrary on most filesystems, so leaving it unsorted means the order changes between polls for no visible reason.

24.10 **GCP Access**

GCPAccess is how to authenticate to Google Cloud.

Defined as GCPAccess in `internal/config`.

Key	Type	Required	Default
project	text	no	—
credentials_file	text	no	—
endpoint	text	no	—

24.10.1 **credentials_file**

CredentialsFile is a service account's JSON key.

Empty uses the service account attached to the VM, GKE pod or Cloud Run service Perfuse runs on (the metadata server), unless Endpoint names an emulator.

24.11 **GCS Access**

GCSAccess is a Cloud Storage bucket.

Defined as GCSAccess in `internal/config`.

Key	Type	Required	Default
bucket	text	yes	—

24.12 IBMMQ Connection

IBMMQConnection is a queue reached through the queue manager's messaging REST API.

Defined as IBMMQConnection in `internal/config`.

Key	Type	Required	Default
<code>url</code>	text	yes	—
<code>queue_manager</code>	text	yes	—
<code>queue</code>	text	yes	—
<code>username</code>	text	no	—
<code>password</code>	text	no	—
<code>tls</code>	tlsconf.Settings (a block, omit to leave off)	no	—
<code>api_version</code>	text	no	—
<code>timeout</code>	duration, such as 30s or 5m	no	30s

24.12.1 username

Username and Password are an mqweb user with the MQWebUser role; the password may be `${ENV}`.

Leave both empty when a client certificate (`tls.cert_file`) authenticates.

24.12.2 password

Username and Password are an mqweb user with the MQWebUser role; the password may be `${ENV}`.

Leave both empty when a client certificate (`tls.cert_file`) authenticates.

24.12.3 timeout

Timeout bounds each request.

Defaults to 30s.

24.13 Kafka SASL

KafkaSASL authenticates to a Kafka cluster.

Defined as KafkaSASL in `internal/config`.

Key	Type	Required	Default
mechanism	text	yes	—
username	text	yes	—
password	text	yes	—

24.13.1 mechanism

Mechanism is "plain", "scram-sha-256" or "scram-sha-512".

PLAIN sends the password readable on the wire, so it belongs with TLS. SCRAM does not, which is why it is worth naming the difference here rather than treating the three as interchangeable.

24.13.2 password

Password is the SASL password.

Supports `#{ENV}` references like every other secret here, so a cluster credential need not be written into the channel file.

24.14 MQTT Connection

MQTTConnection is how to reach an MQTT 3.1.1 broker: Mosquitto, EMQX, HiveMQ, AWS IoT Core, Azure Event Grid.

Defined as MQTTConnection in `internal/config`.

Key	Type	Required	Default
addr	text	yes	—
client_id	text	no	—
username	text	no	—
password	text	no	—
tls	tlsconf.Settings (a block, omit to leave off)	no	—
qos	whole number (a block, omit to leave off)	no	—
timeout	duration, such as 30s or 5m	no	30s

24.14.1 client_id

ClientID names the session.

A source keeps its messages across reconnects only under a fixed client id; it defaults to perfuse-`<channel>`.

24.14.2 **username**

Username and Password authenticate; either may be empty.

The password may be `${ENV}`.

24.14.3 **password**

Username and Password authenticate; either may be empty.

The password may be `${ENV}`.

24.14.4 **timeout**

Timeout bounds connecting and each wait for the broker.

Defaults to 30s.

24.15 **NCPDP Options**

NCPDPOptions are the settings that only apply to a pharmacy claim channel.

Defined as NCPDP0ptions in `internal/config`.

Key	Type	Required	Default
transformations	list of <code>ncdpd.Step</code>	no	—

24.15.1 **transformations**

Transformations are declarative changes applied to each transmission before delivery.

The same vocabulary as every other format - set, copy, clear, map, replace, trim, case, each with an optional condition - because the step engine in `internal/steps` is generic. What differs is the notation: a path is written D1 or 07-D7, using the standard's own two-character field identifiers.

24.16 **Queue Config**

QueueConfig turns on durable queueing for a destination.

Without it, a destination that is down for two minutes loses everything sent during those two minutes: retries happen in memory and when they run out the message is recorded as failed and gone. With it, the message is on disk and keeps its place until the receiver comes back.

It is off by default, which is a deliberate choice rather than caution. A queue changes what an acknowledgement means and it changes ordering behaviour during an outage, and both of those are things an operator should opt into knowingly rather than discover.

Defined as QueueConfig in internal/config.

Key	Type	Required	Default
enabled	true or false	no	—
max_attempts	whole number	no	—
backoff	duration, such as 30s or 5m	no	—
max_backoff	duration, such as 30s or 5m	no	—
max_depth	whole number	no	—
retain_hours	whole number	no	—

24.16.1 max_attempts

MaxAttempts is how many times the queue tries before marking a message failed and moving on to the next one.

Zero means keep trying indefinitely, which is legitimate for a feed that must not lose anything but should be chosen rather than inherited.

24.16.2 max_backoff

MaxBackoff caps the growing delay.

The default is a minute rather than something larger because a receiver that comes back after a brief outage should be found quickly: an hour-long ceiling turns a two-minute blip into an hour of backlog.

24.16.3 max_depth

MaxDepth refuses new messages once this many are waiting, so a receiver that has been down for a week cannot fill the disk.

Zero means no limit.

When the limit is reached the message is failed rather than queued, and the sender is told so — which is the honest answer, because we are no longer promising to deliver it.

24.16.4 retain_hours

RetainHours is how long delivered and abandoned items are kept before being purged.

They are kept at all so somebody watching a backlog drain can see it happening, and so an abandoned message can be found afterwards.

24.17 Retry

Retry controls redelivery after a failure.

Defined as `Retry` in `internal/config`.

Key	Type	Required	Default
<code>attempts</code>	whole number	no	—
<code>backoff</code>	duration, such as 30s or 5m	no	—
<code>max_backoff</code>	duration, such as 30s or 5m	no	—

24.17.1 attempts

Attempts is the total number of tries, including the first.

Zero means the default.

24.17.2 backoff

Backoff is the wait before the second attempt.

It doubles each time, up to `MaxBackoff`.

24.18 Script Options

`ScriptOptions` configures an NCPDP SCRIPT channel.

Why SCRIPT gets a block when its filter is top-level

The channel filter for every data type is the top-level filter key, because `internal/expr` is generic over the message type: one key, compiled against whichever tree or claim the channel carries. Steps cannot be arranged that way. A v2 step addresses a segment and a field, an X12 step addresses a fixed-width element, and the types are different, so one yaml key would have to hold whichever the data type implied - which is a field that means different things depending on another field, and the failure mode is a step that silently does nothing.

So this follows hl7v3, x12, ncpdp and delimited: the steps live in the format's own block, and the top-level transformations key is refused on a SCRIPT channel rather than accepted and ignored.

Defined as `ScriptOptions` in `internal/config`.

Key	Type	Required	Default
<code>transformations</code>	list of <code>hl7v3.Step</code>	no	—

24.18.1 transformations

Transformations are the declarative steps, addressing the prescription as an XML tree.

hl7v3.Step rather than a type of its own, and that is not laziness. A prescription and a v3 document are both XML, the path grammar is the same one, and internal/eprescribe already borrows hl7v3.Path for its filter with a comment saying so - v3 and CDA already use it, and a third copy would be a third set of answers to what `//a/b[2]@c` means.

The steps that carry v3-only meaning are refused at load rather than being silently available. nullflavor is the one: it writes the v3 attribute stating why a value is absent, which a prescription has no equivalent of, and a step that wrote it would produce a document the receiver does not understand.

24.19 Scripts

Scripts holds the JavaScript attached to a channel.

Defined as `Scripts` in `internal/config`.

Key	Type	Required	Default
filter	text	no	—
transformer	text	no	—
preprocessor	text	no	—
deploy	text	no	—
undeploy	text	no	—
postprocessor	text	no	—
language	text	no	—
timeout	duration, such as 30s or 5m	no	—
include	list of text	no	—
file_roots	list of text	no	—
allow	list of text	no	—

24.19.1 filter

Filter runs after the expression filter, if any, and must return a boolean.

A Mirth filter script drops in here unchanged.

24.19.2 **transformer**

Transformer runs after the declarative steps and may modify the message.

A Mirth transformer script drops in here unchanged.

24.19.3 **preprocessor**

Preprocessor runs before the message is parsed, reads the text as "message" and returns the text to parse.

A Mirth preprocessor drops in here unchanged.

This is the only place a message that does not parse can be repaired. Sites use it for a stray character from a serial gateway, a segment terminator that arrived as a line feed, or a vendor padding a field with something illegal for its position. Without it, Perfuse rejects traffic that Mirth accepts today.

24.19.4 **deploy**

Deploy runs once when the channel starts, before it accepts anything.

This is where sites warm a lookup cache or check that something they depend on is reachable. It runs before the source binds, so a deploy script that fails stops the channel from starting rather than letting it accept messages it cannot handle.

24.19.5 **undeploy**

Undeploy runs once when the channel stops.

After the source has stopped accepting and after in-flight messages have finished, so it sees a channel that is genuinely idle. Its failure cannot stop a stop: a channel that could not be shut down because a cleanup script threw would be worse than one that logged and closed.

24.19.6 **postprocessor**

Postprocessor runs after the message has been handled and cannot change it.

The message and what happened to it are visible, so it can notify, count or record. It cannot alter what was sent: by the time it runs the sending has happened, and a script that appeared to change a delivered message would be lying about it.

24.19.7 **language**

Language is which scripting language these scripts are written in: javascript or lua.

Why one setting for the whole block rather than one per script

A channel's scripts share the channel map and the connector map, and they are read together by whoever maintains them. A filter in Lua beside a transformer in JavaScript would mean two languages in

one review, sharing state through a map whose values would have to mean the same in both. Sites that want that can split the channel.

Why the default is javascript

Every script written before this setting existed is JavaScript and says nothing about its language. Making lua the default would silently reinterpret all of them; requiring the field would refuse them. Neither is acceptable for a setting added later.

24.19.8 **timeout**

Timeout bounds each script.

Zero means five seconds.

There is deliberately no way to disable it. Mirth allows a transformer to loop forever, which stops the channel, and an interface that stops accepting admissions because of a typo is not something to offer as an option.

24.19.9 **include**

Include names files of shared script source made available to every script on this channel, in whichever language scripts.language names.

This is Mirth's code templates. A site with forty channels usually has one library every channel calls, and without it a migration means copying that library into forty files - forty places to fix the next time it changes.

Paths are relative to the channel file, so a library beside the channels travels with them. It is a list because the alternative is one enormous file, and libraries tend to divide by subject.

The files have to be in the channel's language. This was JavaScript-only for a while, and not by refusal: a Lua channel read the files, refused a missing one, and then the runtime installed none of it, so every call into the library failed on the first message with a nil global - the exact error the refusal above exists to avoid. Both languages install it now, and a library that does not parse is refused at load naming the file.

24.19.10 **file_roots**

FileRoots are the directories FileUtil may read and write.

Required whenever "file" is allowed. Granting file access without naming directories used to mean the whole filesystem: a script could read Perfuse's own database, which holds password hashes, the LDAP service account password and the OIDC client secret. Creating a channel needs only the editor role, so the escalation completed the moment an administrator started it.

24.19.11 **allow**

Allow grants capabilities a script would otherwise be refused: "file", "database", "route".

Mirth grants all of these to every script; requiring them to be named means a transformer cannot quietly start reading the filesystem after a copy-paste.

24.20 Shadow

Shadow runs a candidate version of a channel beside the live one and reports where they differ.

This is the answer to the question that makes interface work frightening: how do you know a change is safe? Not by reading it. A transformation is a program operating on messages whose variety nobody has catalogued, and the case that breaks is always the one nobody thought of - the patient with two identifiers, the result with an empty units field, the A08 that arrives before its A01.

A channel test proves the cases somebody thought of. Shadow mode proves the cases that actually arrive, against real traffic, before the change carries any of it.

Two properties make it safe, and both are structural rather than configurable:

- The shadow never delivers. Its destinations are not constructed, so there is no code path from a shadow channel to a receiver, and no setting can create one by mistake. - The shadow never affects the live channel. It runs after the live message has been delivered and acknowledged, and a shadow that panics, throws or hangs is recorded as a shadow failure and nothing else.

Defined as Shadow in `internal/config`.

Key	Type	Required	Default
channel	text	yes	—
sample	number	no	1
compare	list of text	no	—
ignore	list of text	no	—
max_differences	whole number	no	100
timeout	duration, such as 30s or 5m	no	5s

24.20.1 channel

Channel is the path to the candidate channel file.

A separate file rather than an inline block, on purpose: the candidate is meant to become the live channel, so it should be the artefact that gets reviewed, committed and promoted, not a fragment that has to be extracted.

24.20.2 sample

Sample is the fraction of messages to shadow, from 0 to 1. Defaults to 1.

Worth lowering only on a busy feed, and worth understanding before doing so: sampling reduces cost and reduces confidence in exactly the same proportion, and the message that would have shown the difference is the unusual one.

24.20.3 **compare**

Compare limits the comparison to these field paths.

Empty compares everything.

24.20.4 **ignore**

Ignore excludes field paths from the comparison.

Needed in practice, because a channel that stamps a timestamp or a sequence number differs on every single message and would report a difference rate of 100% while telling you nothing.

24.20.5 **max_differences**

MaxDifferences bounds how many differing messages are kept for inspection.

Defaults to 100.

24.20.6 **timeout**

Timeout bounds one shadow run.

Defaults to 5s.

Separate from the channel's own timeout, and shorter. A candidate with a runaway script must not accumulate goroutines behind the live traffic.

25 How Each Component Works

Perfuse is built from small packages, each with one job. This chapter is each one's own account of what it does and why it is built the way it is, taken from the source. Where a decision has a reason that is not obvious, the reason is here rather than in a commit message nobody will find.

25.1 hl7

Package hl7 reads HL7 version 2 messages.

It has no dependencies outside the standard library, and a test enforces that. It does no network I/O, writes no files and starts no goroutines, so it is safe to use from anywhere.

Reading a message

```
msg, err := hl7.Parse(raw) if err != nil { return err } mrn := msg.MustGet("PID-3.1") name :=  
msg.MustGet("PID-5.1") msgType, event, _ := msg.Type() // "ADT", "A01"
```

Paths are the notation people already use when they talk about HL7: segment, field, then optional component and subcomponent, as in PID-5.1 or OBX-5.2.3. Repeats are addressed with square brackets, as in PID-3[2].1.

Get returns an error for a path it cannot parse; MustGet returns the empty string instead. Both are safe on absent segments and fields, because a real feed is full of both and a parser that panics on missing data is useless for the job. Absence is not an error — a message that simply does not carry PID-8 yields "" rather than a failure.

Absent and empty

HL7 distinguishes a field that was not sent from one sent deliberately empty, and clinically those can mean different things - "no allergies recorded" is not "no known allergies". Value.Exists reports the difference where callers need it, while String flattens both to "" for the common case.

Copying, and the one way to get this wrong

Parse copies its input, so the returned Message stays valid however the caller reuses its buffer. That costs about 3% and one allocation per message.

WithZeroCopy skips the copy and keeps a reference to the caller's slice instead. It is faster, and it makes the Message unsafe to use after that slice is written again. Reading into a reused buffer is the normal, efficient way to read from a network in Go, so the trap catches careful code, and the symptom is silently wrong field values rather than an error - one patient's data under another patient's name. Use it for input that is immutable or outlives the Message, and not otherwise.

Acknowledgements

AckFor builds an ACK or NAK for a message, including for one that failed to parse, which is the case that matters most: a sender needs a reply even when what it sent was not a message. See AckOptions.

What this package does not do

It does not write or modify messages. It has no notion of message structure beyond segments, so it will not tell you that a PV1 belongs to a particular visit group. It does not validate against a version's tables. Those are deliberate omissions rather than oversights: this is an indexer for real traffic, which is frequently not standard-conformant and still has to be routed.

Source: github.com/biodream-llc/perfuse/hl7

25.2 admit

Package admit bounds how much work is in flight at once.

The engine spawns a goroutine per destination per message. Goroutines are cheap; what they hold is not. Each outbound delivery holds a socket at the far end and a file descriptor here, and it holds them for the whole retry budget - with the defaults, five attempts at a thirty second timeout plus fifteen seconds of backoff, so nearly three minutes. A few hundred senders against one receiver that has gone quiet is therefore a few hundred descriptors held for minutes, and the process runs out.

Running out is the failure worth preventing, because it does not surface where it happened. It surfaces as "too many open files" somewhere unrelated: the database cannot open its journal, the interface stops accepting, a log cannot be written. The engine looks broken in three places at once and the cause is a partner system that stopped answering.

Two limits, and the second matters more than the first:

A total, so the process stays inside its descriptor budget.

A limit per destination, so one receiver cannot consume the total. Without it a single hung partner takes down every other channel, which is worse than having no limit at all: the blast radius grows from one interface to all of them.

Source: github.com/biodream-llc/perfuse/internal/admit

25.3 alerts

Package alerts watches the metrics and says when something is wrong.

The point is that nobody should have to be looking at the dashboard. "Failed transactions remain queued without timely investigation" is a documented operational failure in this class of software, and the reason is always the same: the information was on a screen nobody was in front of at four in the morning.

Three decisions shape it.

An alert must clear itself. A notification that fires and never resolves trains people to ignore notifications. Every rule here is evaluated against current state, so a receiver coming back closes the alert it opened.

Alerts are debounced by duration, not by count. "Error rate above 5% right now" fires on a single unlucky message during a quiet hour. "Above 5% for two minutes" does not. Every rule carries a For duration and nothing fires until the condition has held that long.

What is sent contains no patient data. An alert carries a channel name, a destination name, a number and a threshold. Those come from configuration and from counting. A webhook goes to a chat room, a chat room has a scrollbar, and a scrollbar is not somewhere a patient identifier should end up.

Source: github.com/biodream-llc/perfuse/internal/alerts

25.4 amqp

Package amqp is an AMQP 1.0 client: one connection, one session, sending and receiving links, settled after the receiver says so.

AMQP 1.0 is what Azure Service Bus and Event Hubs speak, what RabbitMQ 4 speaks natively, and what ActiveMQ and Artemis speak beside STOMP. Written here for the reason the STOMP client and the AWS signer are: a full client library would be most of the binary's dependency list for four performatives that matter.

What it does: SASL PLAIN or ANONYMOUS (Service Bus takes a shared access key name and key as PLAIN), TLS, open/begin/attach, credit-based flow, a transfer split into frames when it exceeds the peer's maximum frame size, and dispositions - accepted, released, rejected. The sender waits for the broker's disposition before reporting success, so "sent" means the broker took custody. Empty frames keep the connection alive inside the peer's idle timeout, which Service Bus enforces.

What it does not: transactions, multiple sessions, link recovery and message annotations beyond reading them.

Source: github.com/biodream-llc/perfuse/internal/amqp

25.5 analyze

Package analyze turns a parsed Mirth channel into something a human can act on: a plain description of what the channel does, and an honest list of what will not survive a migration.

The design rule here is that a finding must be specific enough to act on. "This channel uses JavaScript" is not a finding. "Destination 1 calls java.util.Date, which has no equivalent outside the JVM" is.

Source: github.com/biodream-llc/perfuse/internal/analyze

25.6 api

Package api serves the HTTP interface used by the web front end.

The important design constraint: channels are read from and written to files on disk. The API is an editor for those files, not a different storage system that happens to export YAML. A channel created in the browser and one written by hand are the same artifact, which is what makes "export to share" trivial and keeps git as the history of record.

Source: github.com/biodream-llc/perfuse/internal/api

25.7 attach

Package attach moves large payloads out of a message and puts them back.

An HL7 message carrying a scanned report or a PDF is ordinary in document workflows, and the payload is routinely hundreds of times larger than the message around it. Storing that inline means the message store grows at the rate of the documents rather than the traffic, a message queued for five destinations is copied five times, and the interface has to stream megabytes to show somebody a patient name.

So the payload is extracted, stored once by content, and replaced in the message with a token. Mirth calls these attachment handlers and it is a documented migration feature, which is why the token format here is designed to be recognisable rather than clever.

What this package does not decide

It does no storage and no I/O. Extraction produces the rewritten message and the payloads; a Store somewhere else decides where those go. That separation is what makes the extraction testable without a database, and it is also what lets the same logic run in the WASM playground where there is no database at all.

Source: github.com/biodream-llc/perfuse/internal/attach

25.8 authlimit

Package authlimit throttles failed authentication attempts by source address.

It exists because two listeners needed the same thing and one of them had nothing. The FHIR server got a limiter when it got authentication; the console login had none at all, and a measurement showed twenty password guesses a second sustained against the administrator account with no delay and no lockout. At that rate a weak password is gone overnight and the only trace is a log full of identical warnings.

Deliberately not a general-purpose rate limiter. It throttles failures, not requests: a busy legitimate client is never affected, and the only thing that accumulates is being wrong.

Source: github.com/biodream-llc/perfuse/internal/authlimit

25.9 awsmmsg

Package awsmmsg speaks to Amazon SQS and SNS with the standard library and the shared Signature Version 4 signer.

The same argument as s3put: the AWS SDK would double the size of the bill of materials for four API calls. SQS uses its JSON protocol (AWS JSON 1.0, X-Amz-Target AmazonSQS.<Action>), which AWS documents as the current one; SNS uses its query protocol, which is the only one it has. Both work against LocalStack and VPC endpoints through an endpoint override.

Source: github.com/biodream-llc/perfuse/internal/awsmmsg

25.10 awsv4

Package awsv4 signs requests to AWS with Signature Version 4, using only the standard library.

It began inside s3put. SQS, SNS and reading from S3 need the same signature with a different service name and with a query string, and a second copy of the signing code would be a second place for the one bug that matters - a signature mismatch reads exactly like wrong credentials. So it lives here, checked against AWS's published test vectors, and every AWS connector uses it.

Credentials are static keys or the standard environment variables. There is deliberately no instance-metadata credential chain: it means IMDS calls, caching and refresh, which is where a small implementation stops being small, and Perfuse refuses metadata addresses as destinations by default anyway.

Source: github.com/biodream-llc/perfuse/internal/awsv4

25.11 azblob

Package azblob reads and writes Azure Blob Storage with the standard library: Shared Key or SAS authorisation over plain HTTP calls.

The same reasoning as s3put: the Azure SDK is large and the operations an integration engine needs are five - put, get, list, delete, and setting the access tier. Shared Key signing is specified exactly ("Authorize with Shared Key", Azure Storage REST), and Azurite, Microsoft's emulator, checks it the way the service does.

Source: github.com/biodream-llc/perfuse/internal/azblob

25.12 branding

Package branding stores a customer's logo and product name so an installation can present itself as their own product rather than as Perfuse.

Why the logo is not a setting

Everything else an operator can change is a scalar in the settings file. A logo is a few hundred kilobytes of binary, and putting it there would mean base64 in a YAML file that a person is meant to be able to read and hand-edit. It lives beside that file instead, and only its presence is visible through the settings surface.

The part that matters: an uploaded image is untrusted input

A logo is uploaded by an administrator and then served to every browser that loads the console, including on the sign-in page before anyone has authenticated. That makes it the most widely distributed piece of attacker-supplied content in the product, so it gets treated accordingly:

- The declared content type is ignored. The bytes are sniffed, and the result must be in an allowlist. A file called logo.png containing HTML is HTML. - SVG is accepted, because that is what companies actually have, but it is an executable document format: it can carry script elements, event handler attributes, external references and embedded foreign objects. It is rewritten to remove all of those before it is stored, and the rewrite is verified by planting each attack in a test. - It is served with a content security policy that permits nothing, with sniffing disabled, and as an attachment-free inline image only. Even a sanitiser bug should not become script execution on the console's own origin. - The uploaded filename is never used to build a path. The stored name is derived from the sniffed type, so a name like ../../etc/perfuse cannot escape anywhere.

Source: github.com/biodream-llc/perfuse/internal/branding

25.13 cda

Package cda reads clinical documents.

The design starts from how these documents actually travel, which is not as files. A discharge summary leaves an EHR inside an HL7 v2 MDM^T02 message, with the whole document base64-encoded in OBX-5 and described by a TXA segment. That is thousands of messages a day in a mid-sized hospital, and it is why this package sits next to an HL7 v2 parser rather than in a document toolkit: the v2 half is already here, so the pipeline from "a message arrived" to "a FHIR DocumentReference exists" is short.

Three things are deliberately in scope and one is deliberately not.

In scope: reading a document and saying what it contains in plain terms; checking that the human-readable narrative and the coded entries agree; and converting to FHIR. The narrative check is the unusual one. Every C-CDA section carries the same information twice, once as text for a clinician and once as codes for a machine, and they are supposed to match. In practice they often do not, and nothing checks it - not the certification tooling, not the engines. When the text says a patient is allergic to penicillin and the coded entry says something else, one of the two readers acts on the wrong information. It is a patient safety defect that is mechanically detectable, so it is detected here.

Not in scope: generating conformant C-CDA. Writing one means satisfying several hundred template rules and a Schematron, and producing documents that are 95% right would fail certification in ways

that are miserable to debug. Reading, checking and converting is most of the value for a fraction of the risk.

Source: github.com/biodream-llc/perfuse/internal/cda

25.14 **channeltest**

Package `channeltest` runs tests against a channel definition.

A channel is a program. It filters, rewrites and routes clinical messages, and it is edited under pressure by people who cannot try it against production. The absence of a way to test one is the single biggest reason interface work is frightening, and "I changed the transformer and nothing looked different in the message browser" is not a test.

The central decision is that a test runs the message through the real transformation path — the same filter, the same declarative steps, the same script engine, in the same order — with only the transport replaced. A test that exercised a reimplementations would pass while the channel failed, which is worse than having no test at all because it actively misleads.

Source: github.com/biodream-llc/perfuse/internal/channeltest

25.15 **cms0057**

Package `cms0057` holds the payer-side pieces of the CMS Interoperability and Prior Authorization final rule (CMS-0057-F) that are about data rather than transport: adjudicated claims as CARIN Blue Button `ExplanationOfBenefit` resources, prior authorisations as Da Vinci PDex `ExplanationOfBenefit` resources, HREx member matching for the Payer-to-Payer API, the member list behind the Provider Access API, and the prior authorisation metrics a payer has to publish every year.

The FHIR endpoints that serve these live in `fhirserver`; this package decides what goes in them.

Drugs are out of scope throughout, as they are in the rule's prior authorisation provisions: no pharmacy `ExplanationOfBenefit`, no formulary, no NCPDP.

Source: github.com/biodream-llc/perfuse/internal/cms0057

25.16 **codeset**

Package `codeset` holds mapping tables that outlive the channel that first needed one.

The domain tax is largely re-deriving a mapping somebody already derived. A site's sex codes, patient classes, order statuses and facility identifiers get translated the same way in every channel that touches them, and today each of those translations is a separate table inside a separate channel. Forty channels means forty copies and forty places to fix the next time the receiving system changes.

So a table becomes a named object referenced by name. That much is obvious and it is the smaller half of the value.

The half nobody builds

The larger half is provenance. Most of what makes an interface hard to maintain is not the mapping - it is that nobody knows **why**. "We strip leading zeros because the lab's MRN column is numeric" is knowledge that exists in one person's head and leaves the building when they retire, and the mapping that remains becomes something nobody dares change and nobody dares delete.

So every table records who decided and when, every entry may record its own reason, and both survive into the file that goes into version control. A mapping with a reason can be argued with. A mapping without one becomes superstition.

Why the file is separate from the channel

Because it is shared. A table inside a channel is a table only that channel can use, which is the problem. And a shared table needs its own history: "who changed the sex mapping, and when" is a question about the table, not about whichever channel happened to be edited that day.

Source: github.com/biodream-llc/perfuse/internal/codeset

25.17 compliance

Package compliance implements the CMS ADT Event Notification workflow required by 42 CFR 482.24(d) since May 2021.

Hospitals must electronically notify a patient's established providers — primary care, specialists, post-acute care facilities — when the patient is admitted, discharged, or transferred. The regulation exists because patients fall through the cracks during transitions: a PCP who does not know about a discharge cannot schedule the follow-up that prevents a readmission.

This package provides the machinery to:

- Track which providers are established for each patient.
- Route ADT events to the correct providers at the correct priority.
- Record every notification sent, with delivery status, for CMS surveyor audits.
- Generate compliance reports showing notification rates and failures.

Thread safety is required throughout because ADT messages arrive concurrently from multiple channels and the notification pipeline runs in parallel.

Source: github.com/biodream-llc/perfuse/internal/compliance

25.18 config

Package config loads channel definitions from files.

Configuration lives in files rather than a database, which is the main thing Perfuse does differently from Mirth. Mirth keeps channels in its own database, which is why version control is a separate product, why promoting a channel from test to production is difficult, and why nobody can answer "what changed last

Tuesday" without a support call. Files give git, diffs, review, CI and environment promotion with no extra machinery.

Validation is deliberately strict and happens at load: unknown fields are an error, filter expressions are compiled, addresses are checked. A channel that cannot be understood must refuse to start rather than run in a way nobody intended.

Source: github.com/biodream-llc/perfuse/internal/config

25.19 contract

Package contract turns what a feed looks like into what a feed must look like.

A profile answers "what am I receiving". That is useful once. The expensive problem is the day it changes: a vendor upgrade drops a field, a new code appears, a segment starts repeating. None of those is an error - the messages are still valid HL7 and every engine in the world accepts them - so nothing notices until a receiver falls over, usually weeks later, and the investigation starts from the wrong end.

A contract is the profile written down as an expectation and then checked. It is contract testing, which is a solved idea in software generally and almost entirely absent from healthcare integration.

Why the unit is a rate and not a message

The obvious design is to check each message and complain when one fails. That produces a system nobody can use. Real feeds contain a proportion of genuinely odd messages - a test message from the vendor, a patient with no recorded sex, a manual entry - and alerting on each one trains people to ignore the alerts, at which point the feature is worse than nothing because it has consumed the attention it needed.

So an expectation is about a proportion over a window: "PID-3 is populated in at least 99% of messages". One odd message never fires. Five percent of messages losing their medical record number does.

A contract describes what arrives, not what leaves

This is the distinction people get wrong first, including me while integration-testing it. Perfuse stores the message as it arrived - that is deliberate, because the stored message is evidence of what a sender sent - so a contract is checked against the incoming feed and not against the output of the channel's transformations.

That is the right default by a wide margin: the thing that changes without warning is the sending system, and the point of a contract is to notice. But it means an expectation written about a mapped value ("after mapping, only the lab's codes should remain") will fail immediately and confusingly, because the codes it names never appear in the traffic being profiled. Anything about what leaves belongs in a channel test, which runs the real transformations.

Why a violation names the tolerance it broke

"PID-3 failed" tells somebody nothing. "PID-3 was populated in 94.2% of the last 500 messages; the contract requires 99%" tells them how bad it is, how confident to be, and whether it is worth waking somebody. The numbers are the message.

Source: github.com/biodream-llc/perfuse/internal/contract

25.20 crd

Package `crd` is a Da Vinci Coverage Requirements Discovery service (CRD 2.2.1): the payer's side of the CDS Hooks conversation that tells a clinician, while the order is still a draft, whether it is covered, whether it needs prior authorisation, and what documentation the payer will want.

CMS-0057's Prior Authorization API is in practice three Da Vinci guides together: CRD to say what is required, DTR to gather it, PAS to submit it. Perfuse had PAS. This is the first of the other two; DTR's `$questionnaire-package` is on the FHIR endpoint.

The coverage decisions come from a rules file, because they are the payer's policy and change without software releases: a code (or a list of codes) and what applies to it - covered or not, prior authorisation needed or not, documentation, and the DTR questionnaire that collects it. An order matching no rule is answered "conditional" and "indeterminate", which is what CRD says to return when the service cannot decide - never a guess that it is covered.

Responses follow the guide: a `systemAction` updating each order with the coverage-information extension, which the EHR stores on the order and later sends on with the prior authorisation; and a card per order, so a clinician sees it.

Source: github.com/biodream-llc/perfuse/internal/crd

25.21 dashboards

Package `dashboards` defines the dashboards Perfuse ships for each kind of person who watches it, and how a person is given one: by their role, by a directory group the dashboards file maps, or by the view they saved.

Dashboards are made of tiles, and tiles show aggregates only - counts, rates, times, states. None shows a message, a patient or an identifier, so a dashboard can be given to a department manager or put on a wall screen without widening who sees patient data. A tile that needs data Perfuse does not yet record is not faked: the dashboard lists it under `Planned`.

Full flexibility is Grafana's job, not this package's. Every tile that has a metric behind it exports as a Grafana panel over Perfuse's `/metrics`, so a site that wants more than light configuration takes the dashboard there.

Source: github.com/biodream-llc/perfuse/internal/dashboards

25.22 dbtime

Package dbtime formats timestamps so that sorting them as text sorts them as time.

Every store in Perfuse keeps timestamps as TEXT in SQLite and relies on string comparison for ordering and for range queries: ORDER BY received_at DESC, WHERE expires_at < ?, WHERE at >= ?. That is a reasonable design - it keeps a database somebody can read with the sqlite3 shell - but it only works if the text form is fixed width.

time.RFC3339Nano is not. Its documentation says it removes trailing zeros from the seconds field, so the same clock produces strings of different lengths:

20:56:33.12345Z <- 123450 microseconds, trailing zero dropped 20:56:33.123456Z <- 123456 microseconds

Compared as text, the first is **greater**, because after the shared prefix "12345" it has 'Z' where the second has '6', and 'Z' sorts above every digit. So the earlier instant sorts later. Roughly one adjacent pair in ten is affected, which is why it presented as an intermittent test failure rather than as an obviously broken feature.

What it actually broke: the audit log and the message browser both order by a timestamp and can therefore list entries from the same second in the wrong order. An audit trail is a compliance artefact and a message browser is how somebody reconstructs what happened, so in both cases the order is the point.

Layout below uses .000000000 rather than .999999999. Both print nanoseconds; the difference is that zeros are significant in the first and dropped in the second.

Source: github.com/biodream-llc/perfuse/internal/dbtime

25.23 delimited

Package delimited parses records separated and split by configured characters.

This is Mirth's Delimited data type: CSV, tab-separated, pipe-separated, fixed-column files, and the long tail of formats a laboratory analyser or a bureau service emits because somebody wrote it in 1994 and it works.

Deliberately not a general CSV library. What matters here is what a healthcare feed does that a well-formed CSV does not: a header row that may or may not be present, a trailing empty line that must not become an empty record, columns referenced by name so a transformation does not break when a supplier inserts one, and the ability to treat a whole file as one message or each row as its own.

The last of those is the decision that shapes everything else. A file of five thousand results is either one message or five thousand, and the two are not interchangeable: one message means one acknowledgement and one failure for the whole file, and five thousand means each row can be filtered, transformed and retried on its own.

Source: github.com/biodream-llc/perfuse/internal/delimited

25.24 dicom

Package dicom reads DICOM data sets.

DICOM is where radiology lives, and a hospital integration engine that cannot speak it is shut out of imaging entirely. Mirth has a DICOM listener and sender, so this is a migration blocker for any site doing modality work.

Scope, stated plainly

This package parses data sets: the file meta information, the elements, the transfer syntax, and the values that matter for routing and identification. It does not decode pixel data, and it never will as far as this package is concerned - an integration engine moves images, it does not display them, and a decoder is a large body of format-specific work whose only purpose here would be to make the binary bigger.

Reading metadata without touching pixels is also what makes it fast on the traffic that matters: a chest CT is hundreds of megabytes of pixels behind two kilobytes of the information anybody routes on.

Why the value representations are handled explicitly

A DICOM element carries a two-letter value representation saying how to read its value, and the rules differ per VR in ways that cannot be guessed: some have a two-byte length and some four, some are padded with a space and some with a null, and numeric strings can hold several values separated by backslashes. Getting one wrong does not produce an error, it produces a plausible-looking wrong value - a patient name read from the middle of a date, or a study identifier off by one byte.

Source: github.com/biodream-llc/perfuse/internal/dicom

25.25 drug

Package drug handles the codes and quantities that identify a medicine.

Split out from the pharmacy standards that carry them, because the same drug code appears in an NCPDP claim, a SCRIPT prescription, an HL7 RXE segment and a FHIR MedicationRequest, and a conversion that lives in one of those places is a conversion the other three do without.

Source: github.com/biodream-llc/perfuse/internal/drug

25.26 dsdr

Package dsdr signs and verifies CDA documents as the HL7 Implementation Guide for CDA Release 2: Digital Signatures and Delegation of Rights, Release 1 (October 2014) describes - the electronic

signature standard HIPAA adopts for claims attachments in CMS-0053-F, 45 CFR 162.2002(e), with compliance from 26 May 2028.

What the guide asks for, and where it is done here:

- The digest is over the whole document, leaving out every `legalAuthenticator` and `authenticator` element (section 3.1.2), so later signers do not break earlier signatures. Expressed in the signature as an XPath Filter 2.0 subtract transform, which any XML-DSig implementation can evaluate. - The signature is XAdES-X-L (3.1.1, ESMD-1 to ESMD-6): the signing certificate, UTC signing time, a role from the NUCC provider taxonomy, a signature purpose from ASTM E1762, the certification path in `CertificateValues` and an OCSP or CRL response in `RevocationValues`, with the time-stamps X-L is built on. - It is carried base64 in `sdhc:signatureText` on the participant who signed, with a plain-text thumbnail (3.3, ESMD-7 to ESMD-16), inside `digitalSignature/authorizedSigner`.

Where the guide is silent, the choices made are named: it does not give a namespace for its own elements (`digitalSignature`, `authorizedSigner`, `SignaturePurpose`), so they are put in `urn:hl7-org:sdhc`; and the purpose is also stated as a XAdES `CommitmentTypeIndication` with the ASTM OID, which XAdES tools read and the guide's own element they do not.

Delegation of rights (a delegated signer with a SAML assertion) is not produced; a signature carrying one is reported as such.

Source: github.com/biodream-llc/perfuse/internal/dsdr

25.27 e4x

Package `e4x` makes Mirth Connect's JavaScript run unchanged.

Mirth scripts are written for Rhino, which implements E4X: XML is a native type with its own syntax. The engine here is `goja`, which does not, and the gap is not only a runtime one. Three pieces of E4X are **syntax**, so `goja`'s parser rejects the file outright before any object model could help:

```
for each (var seg in msg.OBX) { } // for-each, and the descendant operator var z = <ZZZ>
<ZZZ.1>value</ZZZ.1></ZZZ>; // an XML literal var id = node.@root; // an attribute reference
```

This file rewrites those into ordinary ECMAScript that `goja` parses, leaving the semantics to the runtime object in `xml.go`. Everything else about E4X - property access by element name, indexing repetitions, assignment, `toString` - is already valid JavaScript syntax and needs no rewriting.

The rewriting is done with a real lexer rather than regular expressions, because the alternative corrupts working scripts. A pattern for the descendant operator finds one inside `"http://example.org"`, a pattern for XML literals finds one in `"a < b && c > d"`, and a script that silently changes meaning is far worse than one that refuses to load.

Source: github.com/biodream-llc/perfuse/internal/e4x

25.28 egress

Package `egress` decides whether this engine may connect outward to an address.

It exists because a destination can be pointed anywhere, and one combination turns that into a way to read the internal network: a SOAP destination with a response transformer returns the receiver's reply to a script, so an editor can author a channel that fetches an internal URL and writes the body into a log or a message.

Demonstrated rather than theorised: a SOAP destination aimed at a local HTTP server put that server's response body into a channel log line.

The proportionate answer is not to forbid outbound connections - delivering messages to configured endpoints is the entire purpose of the software, and an editor is meant to configure them. It is to refuse the specific addresses that are never a message destination and are always a credential store.

Source: github.com/biodream-llc/perfuse/internal/egress

25.29 engine

Package engine runs configured channels.

A channel is a source, a filter, and a set of destinations. The runtime's job is to be honest about what happened to each message: an acknowledgement should mean what the configuration says it means, a message that is filtered out should be distinguishable from one that was delivered, and a message that could not be delivered anywhere must not be reported as accepted.

Source: github.com/biodream-llc/perfuse/internal/engine

25.30 eprescribe

Package eprescribe reads and writes NCPDP SCRIPT, the standard prescriptions travel in.

Named for what it does rather than for the standard, because internal/script is already the JavaScript engine and two packages called script in one tree is a mistake waiting to be made at an import line.

SCRIPT is XML, unlike the Telecommunication Standard that carries the resulting claim. A prescriber sends a NEWRX to a pharmacy; the pharmacy sends back RXFILL to say what happened, REFREQ to ask for a refill, or RXCHG to ask for a change. Every one of those is a message here.

Where the danger is

Two fields in a prescription are dangerous when misread rather than when missing.

Substitutions is the first. A zero means substitution is permitted and a one means dispense as written. It reads backwards to anybody expecting a flag where one means yes, and getting it wrong dispenses a generic where the prescriber required the brand - or refuses a substitution that was allowed, which is the same error costing the patient money instead. So this package refuses to guess: an absent Substitutions element is an error rather than a default, because both defaults are wrong.

Refill count is the second. A schedule II prescription may not be refilled at all, and a system carrying a refill count across from a non-controlled template produces a valid-looking prescription that a pharmacy

must refuse.

What has not been verified

The message structure follows the published SCRIPT element names. No message from this package has been sent through a real routing network such as Surescripts, and certification by one is a commercial process rather than a technical one. This parses and builds the format; it does not claim certification.

Source: github.com/biodream-llc/perfuse/internal/eprescribe

25.31 equiv

Package equiv proves that two engines agree, or says exactly where they do not.

This is the answer to the only question that actually blocks a migration. An interface team does not refuse a new engine because it lacks features; they refuse because they cannot be sure it will do the same thing to the traffic they already have, and being wrong means lab results stop reaching doctors. No amount of feature comparison answers that. Evidence does.

The intended use is a parallel run. Both engines receive the same production messages; the incumbent keeps delivering and Perfuse delivers to nobody, which is what shadow channels already are. After two weeks the outputs are compared here, and the conversation changes from "trust this" to "here are the three messages out of four hundred thousand where we differed, and why".

Grouping is the whole feature

A naive diff of 400,000 messages produces 400,000 findings and gets closed unread. Almost every real difference has one cause: a date format, a facility code, one transformation step nobody carried across. So differences are grouped by cause - the path in the message and the kind of difference - and reported once with a count. Three thousand differences from one cause is one finding, not three thousand. That single decision is what makes the report readable, and a readable report is the difference between a migration that proceeds and one that stalls.

Values are withheld by default

Real traffic is patient data, so a comparison report is a place PHI could leak into a ticket, an email or a slide. By default this reports paths, kinds and counts - never values. That is enough to find the cause: "MSH-7 differs in 3,000 of 3,000 messages" tells an analyst it is the date format without showing one date.

Values can be enabled deliberately, because eventually somebody does need to see one, and pretending otherwise would only push them to a worse tool. It is an explicit opt-in and the report says on its face that it contains message content.

Source: github.com/biodream-llc/perfuse/internal/equiv

25.32 **expr**

The HL7 v2 binding.

This is the logic that used to live inline in eval.go. It is unchanged in behaviour - deliberately, because the v2 filter is what every HL7 channel runs and the point of the generic grammar was to add formats, not to alter this one.

Source: github.com/biodream-llc/perfuse/internal/expr

25.33 **fdlimit**

Package fdlimit finds out how many file descriptors this process may open, and raises the limit if it can.

This exists because of the shape of the failure it prevents. Every inbound connection is a descriptor, and so is every outbound delivery in flight. A receiver that has stopped answering holds a descriptor for the whole retry budget - with the defaults, five attempts at a thirty second timeout plus fifteen seconds of backoff, so nearly three minutes. A few hundred connections against one hung receiver is therefore a few hundred descriptors held for minutes.

Running out does not produce a tidy error in the place that ran out. It produces "too many open files" somewhere else entirely: the database cannot open its journal, the web interface stops accepting, a log file cannot be written. The engine appears to have broken in three unrelated places at once, and the real cause is a partner system that went quiet. That is the support call this package exists to avoid.

Two things happen at startup. The soft limit is raised to the hard limit, which usually removes the problem outright - the soft default is often a few hundred while the hard limit is tens of thousands. Then whatever is available is reported, so the engine can bound its own concurrency inside it rather than discovering it by hitting it.

Source: github.com/biodream-llc/perfuse/internal/fdlimit

25.34 **fhir**

Package fhir implements FHIR resources, serialisation and validation.

Version handling is the first design decision, because "FHIR" on its own does not identify a wire format. R5 is the latest published release and is the default here. R4 is what the installed base and US regulation actually consume, so it is fully supported rather than treated as legacy. R6 is in ballot at the time of writing and is deliberately not implemented: shipping a guess at an unpublished specification would be worse than not supporting it.

The resource model is R5-shaped. Where R4 differs, the difference is applied when serialising rather than by keeping two parallel sets of structs, which is how a codebase ends up with two subtly divergent definitions of Patient.

Source: github.com/biodream-llc/perfuse/internal/fhir

25.35 fhirserver

Package fhirserver implements a FHIR REST server.

This exists because the useful end of a v2-to-FHIR mapper is somewhere to put the result. Being able to convert a message is a demo; being able to convert it, store it, and then find the patient again by identifier is a product.

It is a real FHIR server in the parts that matter for that: a capability statement, create, read, update, delete, search on the parameters people actually use, and transaction bundles applied atomically. It is not a complete FHIR server — no history, no chained search, no `_include` — and it says so in its own capability statement rather than letting a client discover the gaps.

Source: github.com/biodream-llc/perfuse/internal/fhirserver

25.36 framing

Package framing reads and writes messages on a byte stream that is not MLLP.

Why this exists separately from mllp

MLLP is one framing: 0x0B before, 0x1C 0x0D after. It is what HL7 v2 uses over TCP and Perfuse has always spoken it.

A great deal of equipment does not. A laboratory analyser, a scale, a bedside monitor, an older billing system - each tends to have its own convention, and the conventions are all simple variations on the same three ideas: a delimiter somewhere, a fixed record length, or a length written in front of the payload.

Mirth's TCP connector covers this and Perfuse had nothing, so a site with a socket feed that was not HL7 could not use it at all.

The mistake this package is written to avoid

Reading from a stream and guessing where a message ends is the whole problem. Guess short and a message is cut in half; guess long and two messages arrive as one. Both produce output that often looks plausible, which is worse than an error, so every mode here is explicit about what terminates a message and none of them fall back to another.

A message with no framing at all is a legitimate configuration - some devices open a connection, send one message and close - but it cannot be combined with anything else and cannot be used on a connection that stays open, so it is named and validated rather than treated as a default.

Source: github.com/biodream-llc/perfuse/internal/framing

25.37 **ftpconn**

Package `ftpconn` uploads files over FTP and FTPS using only the standard library.

FTP is old, chatty and not a good protocol. It is also still how a large number of hospital systems accept files, usually because the far end is a laboratory analyser or a bureau service whose software has not been touched since it worked. Refusing to support it does not make those systems modern; it makes Perfuse unusable at those sites.

Implemented here rather than taken as a dependency for the same reason as S3: the eight-dependency bill of materials is a real argument with whoever reviews software before a hospital installs it. FTP's control channel is line-oriented text over TCP, so this is a few hundred lines of net and crypto/tls.

What is supported, and what is refused

Passive mode only. Active mode requires the server to open a connection back to the client, which every hospital firewall built this century blocks, and offering it would mean people configuring it and then debugging a firewall.

Explicit FTPS (AUTH TLS) is supported and is the default when TLS is asked for. Implicit FTPS on port 990 is also supported because some analysers only speak that.

Plain FTP is supported and warns. It sends the password in clear text, and a site that has to use it should be told once rather than left to find out from a packet capture.

Source: github.com/biodream-llc/perfuse/internal/ftpconn

25.38 **gcp**

Package `gcp` talks to Google Cloud Pub/Sub and Cloud Storage over their REST APIs, with Google's OAuth 2.0 done here: a service account key signs a JWT (RS256, crypto/rsa) that is traded for an access token, or, on Google Cloud itself, the metadata server hands one out.

No Google SDK.

Emulators (the Pub/Sub emulator, fake-gcs-server) take no credentials: set Endpoint and leave the credentials empty.

Source: github.com/biodream-llc/perfuse/internal/gcp

25.39 **generate**

Package `generate` makes synthetic HL7 v2 messages.

Two uses. Somebody trying Perfuse needs traffic to look at, and a channel test needs a fixture that is realistic enough to exercise the thing being tested. A generator is a paid extension in Mirth, which tells you people want one.

Everything here is invented. The names come from a fixed list of obviously fictional ones, the identifiers are sequential within a run, and the addresses are made up. That is not a disclaimer, it is the design: a generator that produced plausible-looking real names would eventually put one into a test fixture, a bug report and then a repository.

Realism is spent where it matters for testing and nowhere else. Timestamps carry offsets, repeating fields actually repeat, a name sometimes has a middle initial and sometimes does not, and roughly one message in twelve is deliberately awkward: a missing optional field, an unusual but legal separator, an empty component. Feeding a channel nothing but perfectly formed messages tells you very little.

Source: github.com/biodream-llc/perfuse/internal/generate

25.40 hl7dict

Package hl7dict names the fields of HL7 v2 segments.

This exists for one screen: a message viewer that can tell you PID-5.1 is the patient's family name instead of leaving you to count carets. Anyone who has debugged an interface has done that counting by hand, and it is the single most common source of the off-by-one mistakes that put a value in the wrong field.

It is not the whole standard. HL7 v2 has hundreds of segments and thousands of fields, and the ones here are those that appear in ADT, ORU, ORM and SIU traffic. An unknown field is reported as unknown rather than guessed at.

Source: github.com/biodream-llc/perfuse/internal/hl7dict

25.41 hl7v3

Package hl7v3 reads HL7 version 3 messages.

Version 3 is a different thing from version 2 wearing the same name. Version 2 is a pipe-delimited stream of segments whose meaning is positional; version 3 is XML generated from a reference information model, where meaning is carried by element names and by a small set of datatypes that recur everywhere. Almost nothing transfers between the two beyond the clinical vocabulary.

In practice v3 arrives in one of two shapes. Either it is a CDA document, which is v3-derived and handled by the cda package, or it is an IHE transaction - PIX and PDQ v3 in particular, which is how a great many hospitals resolve patient identifiers across systems. This package targets the second.

Why the datatypes come first

The single most consequential thing about v3 is that an element can be present and explicitly null, with a stated reason. A birth date carrying nullFlavor="ASKU" means the patient was asked and declined to answer. One carrying nullFlavor="NAV" means the value exists but is not available right now, so asking again later may work. One that is simply absent means nobody recorded anything at all.

Collapsing those into an empty string is the mistake this package is arranged to prevent. It is not a cosmetic loss: a downstream system that receives an empty birth date where the source said "patient declined" will usually treat it as missing data to be chased, and a demographic match that should have been left unresolved gets resolved by a human guessing. So every datatype here distinguishes absent, null-with-reason, and present.

Source: github.com/biodream-llc/perfuse/internal/hl7v3

25.42 hl7xml

Package hl7xml converts HL7 v2 messages to and from the XML shape Mirth Connect uses, and back to ER7 encoding.

The shape is not a design choice. Mirth's scripts are written against the output of its own ER7-to-XML serialiser, so they contain expressions like

```
msg['PID']['PID.5']['PID.5.1'].toString()
```

and the promise that an existing script runs unchanged is only kept if the tree it walks has exactly the same element names, nesting and repetition behaviour. Every decision below is therefore "what does Mirth do", not "what would be tidier".

The two that matter most:

Every field is wrapped in numbered component elements even when the field has only one component, so PID-5 with the value "Doe" still produces <PID.5><PID.5.1>Doe</PID.5.1></PID.5>. Scripts index straight to .1 constantly and a tree that put the text on the field element would break nearly all of them.

MSH keeps standard HL7 field numbers, where MSH-1 is the field separator and MSH-9 is the message type. The parser in internal/hl7 indexes MSH with an offset because the separator occupies a position no delimiter splits, and that offset stops here: a script saying MSH.9 means the message type.

Source: github.com/biodream-llc/perfuse/internal/hl7xml

25.43 ibmmq

Package ibmmq sends to and reads from IBM MQ queues through the queue manager's messaging REST API (mqweb), which IBM MQ 9.1.5 and later ship on every platform, z/OS included.

It needs no MQ client library: the native MQI protocol is IBM's and undocumented, and the REST API is the supported way to reach MQ without one. (IBM MQ also speaks AMQP 1.0 on an AMQP channel; Perfuse's amqp connector covers that route.)

Reading is at least once: the first message is browsed, handed to the caller, and removed by its message id only once the caller has handled it. A message that could not be handled stays on the queue, first in line.

Source: github.com/biodream-llc/perfuse/internal/ibmmq

25.44 identity

Finding a message by what you know about it, whatever format it arrived in.

The existing content search matches metadata, offers a substring match against the payload, and can evaluate a filter expression - but the expression is evaluated against HL7 v2 only, and every expression requires knowing the path. So finding one patient's traffic meant knowing that identity lives at PID-3 in an HL7 v2 admission, at Patient.identifier in a FHIR resource, at (0010,0020) in a DICOM instance and at NM109 of an NM1*IL loop in an 837 claim. Four formats, four vocabularies, and during an incident the message being hunted might be in any of them.

This package maps a small set of concepts - who the message is about, which visit, which study, which claim - onto the place each format keeps them, so a search can be for "MRN 12345" rather than for a path. The concepts are deliberately few: these are the things people actually have in their hand when they need to find a message, which is a number off a phone call or a name off a complaint.

Why extraction happens once, at record time

The alternative is parsing payloads at query time, which is what the expression search does. That works and it is honest about being a bounded scan, but it cannot answer "every message about this patient across four feeds and two months" without reading every row. Extracting once into an indexed table turns that into an index lookup.

This is a privacy escalation, and it is optional

Patient names and identifiers currently sit inside an opaque payload column. Indexing them makes them queryable, which is the point, and also makes them enumerable, which is a real change in exposure: a database that could only be grepped can now be asked "list every patient". That is a decision for the site, not for this package, so extraction is switchable and every search through it is recorded in the PHI audit log.

Source: github.com/biodream-llc/perfuse/internal/identity

25.45 kafka

Kafka, as a client rather than as something to replace.

A health system that has an event backbone has a Kafka one, and the systems built in the last decade publish to it rather than to a JMS broker. So this connector exists to make Perfuse the healthcare-aware edge of somebody else's event platform: it reads clinical messages off topics and publishes them onto topics, and it deliberately does not try to be a streaming platform. Windowing, joins, stream processing and schema registries are Kafka's ecosystem and stay there.

Why a dependency, when STOMP and MLLP were written by hand

The stated rule in this repository is that a dependency has to buy capability rather than convenience, and must be pure Go so one static binary still cross-compile to every target. `franz-go` is pure Go and carries no `cgo`, which was checked against all six targets.

Kafka's protocol is not STOMP's. STOMP is a text protocol that fits in a few hundred lines. Kafka is a versioned binary protocol across roughly seventy request types, and the part that matters most here - consumer groups - is a distributed coordination protocol with join, sync, heartbeat and offset-commit phases and a rebalance dance between them. Getting that subtly wrong does not produce an obvious failure: it produces a partition nobody is reading, or two consumers reading the same one. In a clinical feed those are a missing lab result and a duplicated order. That is capability, and it is not worth hand-rolling to save a dependency.

Source: github.com/biodream-llc/perfuse/internal/kafka

25.46 ldap

Package `ldap` speaks enough LDAP to authenticate a person and read their groups.

Hospitals run Active Directory almost universally, and AD speaks LDAP. OpenID Connect is the better answer where it is reachable - see `internal/oidc`, and the note there about Mirth having no single sign-on at any price - but an integration engine often sits on an isolated VLAN where a browser cannot reach an identity provider at all. That is what this is for.

Hand-rolled BER rather than a dependency, for the same reason as DICOM and STOMP: the encoding is small, and what arrives on this socket decides whether somebody gets into a system that moves patient data.

`encoding/asn1` in the standard library is not usable here. It targets DER and rejects the context-specific tags and implicit tagging that LDAP uses throughout, so the parts that matter would need hand-rolling anyway.

Source: github.com/biodream-llc/perfuse/internal/ldap

25.47 mapper

Package `mapper` suggests field mappings between systems with confidence scores.

This is a rules-based intelligent mapping engine. No LLM, no network calls, no external dependencies. It runs offline in the same binary and never sends patient data anywhere.

The engine combines multiple signals - fuzzy string matching on field names, healthcare identifier pattern recognition, code system lookups, and historical acceptance data - into a single confidence score per suggestion. When confidence is below a configurable threshold the mapper abstains rather

than guessing wrong, because a wrong mapping in healthcare can route data to the wrong field and that is a patient-safety defect.

Abstention is the core design decision. A system that always produces an answer trains users to stop checking, and the one time it is wrong nobody notices until a lab result lands in a note field. A system that says "I don't know" forces a human decision exactly when the human is needed.

Source: github.com/biodream-llc/perfuse/internal/mapper

25.48 metrics

Package metrics collects what the engine is doing and keeps enough history to draw it.

An interface engine is watched, not just run. When a ward says results stopped arriving twenty minutes ago, the question is what changed twenty minutes ago, and that cannot be answered by current counters alone. So this keeps series rather than totals, in memory, at a fixed resolution.

Three decisions shape it.

Latency is kept as a histogram rather than a mean. An average delivery time is nearly useless in this domain: the number that matters is whether the slowest few per cent of messages are slow enough to breach a turnaround commitment, and a mean hides exactly that. Buckets are exponential and fixed at build time, so recording a duration is a comparison and an increment with no allocation.

Series are bounded ring buffers, sized when the collector is made. Nothing here grows without limit, because an interface engine runs for months at a time and a metrics system that needs restarting is worse than none.

Cardinality is capped. A label value taken from message data - a sending application, say - can take thousands of values if somebody misconfigures a feed, and unbounded labels are the standard way monitoring takes down the thing it monitors. Past the cap, values collapse into an "other" bucket and the cap is itself reported.

Source: github.com/biodream-llc/perfuse/internal/metrics

25.49 mirth

Java analysis for imported Mirth channels.

Why this exists

Mirth runs Rhino on the JVM, so a Mirth script can call any Java on the classpath, including Mirth's own server classes. Perfuse runs goja and has no Java runtime. It refuses those calls with an error naming the class - but that refusal happens when the script runs, which for a source connector means when the first real message arrives.

So a migration used to go: the import succeeds silently, the channel deploys, and the first message off the wire fails. A migration that looks clean and breaks on first traffic is worse than one that refuses at the

door, because the person who did it has already told their colleagues it worked.

This finds those calls at import time and says what to do about each one.

Why four verdicts and not two

"Uses Java" is one bit of information and the wrong one, because the Java a real Mirth channel uses falls into groups with completely different amounts of work attached. Roughly half of what sites actually write is SimpleDateFormat and HashMap, which JavaScript does natively. Some of it is Mirth's documented script API, which Perfuse already implements under the same names. A little of it wants to start and stop channels, which Perfuse can do over its REST API but does not expose to scripts. And some of it is a vendor jar that is never going to run here.

Reporting those as one number tells a migrator nothing about whether their afternoon or their quarter is at stake.

Source: github.com/biodream-llc/perfuse/internal/mirth

25.50 mirthlive

Package mirthlive finds the running Mirth-family engines that the interoperability tests talk to.

Mirth Connect stopped being open source at 4.6 (March 2025). The sites leaving it go to one of three places: Mirth 4.5.2, the last open release, which they keep running; the Open Integration Engine, the Eclipse-hosted fork; or BridgeLink, Innovar's fork, which numbers its releases by year (26.9.0). A channel Perfuse exports has to load in all of them, and a channel any of them exports has to import here, so every live test runs once per engine that answers.

The engines are found by asking, not configured: `./scripts/interop-up.sh` starts them on fixed ports, and an engine that is not running is reported as skipped rather than silently passing.

`PERFUSE_MIRTH_ENGINES` overrides the list, as `name=url` pairs separated by commas.

Only tests import this. It lives outside a `_test.go` file because two packages share it.

Source: github.com/biodream-llc/perfuse/internal/mirth/mirthlive

25.51 mqtt

Package mqtt is an MQTT 3.1.1 client: connect (with TLS, a username and password, or a client certificate), publish at QoS 0 or 1, subscribe, and acknowledge each received QoS 1 message only once the caller says it was handled.

Written against the OASIS MQTT 3.1.1 standard rather than imported, as Perfuse's other wire protocols are. 3.1.1 is the version every broker speaks: Mosquitto, EMQX, HiveMQ, VerneMQ, RabbitMQ's MQTT plugin, IBM MQ's telemetry channel, AWS IoT Core and Azure Event Grid.

Delivery: a message received at QoS 1 is acknowledged (PUBACK) only when Ack is called. A caller that cannot handle a message closes the connection instead; with a persistent session (CleanSession false)

the broker sends it again, marked as a duplicate, when the client reconnects. MQTT 3.1.1 has no negative acknowledgement, so this is the only way to ask for a message again.

Source: github.com/biodream-llc/perfuse/internal/mqtt

25.52 msgstore

Finding a stored message by what somebody knows about it.

The search a person actually needs is "here is a number off a phone call, show me every message about it". Before this that meant either a substring match over payloads, which is grep and slow, or a filter expression, which requires knowing that identity lives at PID-3 in HL7 v2 and somewhere else entirely in the other three formats on the same server.

Identity is extracted once when the message is recorded and indexed, so this is a lookup rather than a scan. The cost is paid on the recording path, once per message, instead of on every search.

Source: github.com/biodream-llc/perfuse/internal/msgstore

25.53 ncpdp

Writing a field in an NCPDP transmission.

Why a write refuses an ambiguous path

Read returns every value a path addresses, because a filter asking whether any DUR conflict is a therapeutic duplication should see all of them. A write cannot work that way. If D7 addresses the product codes of two claims in one transmission, setting D7 has no single meaning: writing both would change a drug nobody asked about, and writing the first would silently leave the second.

So a write refuses a path that addresses more than one field, and the error names the qualified form that would be unambiguous. This is the same reasoning that gives the transformation steps no remove action: where the honest answer is that the configuration is ambiguous, saying so beats picking one.

Source: github.com/biodream-llc/perfuse/internal/ncdpd

25.54 oidc

Package oidc verifies OpenID Connect identities.

Written against the standard library rather than a JWT dependency. Authentication is the one place where an unaudited dependency is least acceptable, signature verification for the algorithms that matter is about a hundred lines of crypto/rsa and crypto/ecdsa, and Perfuse already carries enough that adding more needs a reason rather than a habit.

The scope is deliberately narrow: verify that an identity provider vouched for a person, and report what it said about them. Everything else - what they may do, how long the session lasts - stays with Perfuse,

because an identity provider knows who somebody is and not what they are allowed to do in an interface engine.

Source: github.com/biodream-llc/perfuse/internal/oidc

25.55 parity

Package parity compares what Perfuse produces against what the engine being replaced produced.

Why this exists

Perfuse can import a Mirth channel and can trace what it would do with a message. Neither of those answers the question somebody actually has before moving a live feed, which is: will this produce the same output as the thing that has been running for six years?

No amount of design quality answers that. The only honest answer is evidence, and the evidence has to come from the old engine's own output on the site's own traffic.

Why the pairs have to be supplied

This cannot reach into Mirth, and pretending otherwise would be the worst kind of feature. What it takes is pairs: a message as it arrived, and the message the old engine produced from it. Both are in Mirth's own message store and can be exported from its message browser, which is a thing an administrator already knows how to do.

That constraint is stated rather than hidden. A tool that claimed to verify a migration without ever seeing the old engine's output would be verifying nothing.

Why differences are grouped rather than listed

A run over fifty thousand messages that reports four thousand differences has told somebody nothing they can act on. The same run reporting that PID-8 differs in every message because the old engine mapped M to Male has told them exactly one thing to decide about.

So findings are grouped by field, with a count, an example, and the distinct value pairs seen. One decision per finding is the unit somebody can work through.

Source: github.com/biodream-llc/perfuse/internal/parity

25.56 pas278

Package pas278 maps between Da Vinci prior authorisation and the X12 278 transaction.

Why this exists

CMS-0057-F requires payers to expose a FHIR prior authorisation API by 1 January 2027. It does not replace the X12 278, and it was never going to: a payer's utilisation management system decides authorisations and speaks 278. So a payer meeting the rule is translating FHIR to 278 and back, and a provider sending 278 today will be sending FHIR tomorrow to the same payers who still accept both.

That translation is the thing worth owning. An engine that holds both models and cannot convert between them leaves the hardest part of the work - and the part where the mistakes are expensive - to whoever is integrating it.

The rules this package follows, stated rather than implied

Written down for the same reason v2fhir writes its own down: mapping is where interoperability projects fail, and the failures are silent.

- **A decision is never upgraded.** Partial certification does not become approval, and a request the payer refused to consider does not become a denial. Both of those collapses are available in the obvious mapping and both are wrong in the direction that costs somebody weeks: an appeal filed against a decision nobody made, or six sessions booked against three approved. - **Nothing is invented to fill a required field.** Where the target format demands something the source does not carry, the mapping reports it as a note rather than supplying a plausible default. A fabricated authorisation number is worse than an absent one. - **Every judgement is reported.** Notes come back with the result, so a person can read the handful that needed a decision rather than the whole output. - **Round trips are tested, not assumed.** A mapping that loses a field in one direction is found by converting back and comparing, which is the only way to see a field nobody thought to assert on.

Source: github.com/biodream-llc/perfuse/internal/pas278

25.57 pdf

Package pdf writes simple text documents, using only the standard library.

Mirth's Document Writer renders HTML to PDF, which needs a layout engine. That is not what this is. This produces a plain, paginated text document with a monospaced font, and it exists because the actual requirement at almost every site that uses that connector is "put this report on a share so somebody can print it" - a discharge summary, a result report, a daily reconciliation. Those are text.

Refusing the feature entirely would leave a Tier 2 migration blocker in place. Pulling in an HTML rendering engine would cost more than the whole rest of the binary and bring a CVE stream with it, which would undo the argument the bill of materials makes. So: text, honestly described as text, and a site that genuinely needs typeset output is better served by a file destination and a tool built for it.

Why write a PDF at all rather than a text file

Because the receiving end is usually a person with a printer and a Windows share, and a .txt file printed from a share loses its pagination, its margins and often its line endings. A PDF prints the same everywhere, which is the entire reason anybody asked for one.

Source: github.com/biodream-llc/perfuse/internal/pdf

25.58 priorauth

Package priorauth implements the provider-side prior authorization workflow required by CMS-0057. It submits prior auth requests to payer FHIR APIs, tracks responses, and converts between internal types and FHIR resources.

Source: github.com/biodream-llc/perfuse/internal/priorauth

25.59 publichealth

Package publichealth implements bidirectional public health reporting: electronic case reporting (eCR), electronic lab reporting (ELR), and immunization registry queries.

Every hospital is legally required to report certain conditions — tuberculosis, measles, hepatitis, COVID-19, sexually transmitted infections, foodborne illness — to state and local public health agencies. The reporting happens through three distinct interfaces:

- ELR: HL7 2.5.1 ORU^R01 messages carrying lab results to public health agencies (elr.go). - eCR: FHIR-based electronic initial case reports (eICR) when a reportable condition is diagnosed. - Immunization registry: VXU^V04 to report vaccinations administered, and VXQ^V01 to query a patient's immunization history.

The package is bidirectional: it reports outward (ELR, eCR, VXU) and queries back (VXQ/VXR). All operations are real-time rather than nightly batch.

Thread safety: SubmissionLog is safe for concurrent use. The builder functions are pure and safe to call from any goroutine.

Source: github.com/biodream-llc/perfuse/internal/publichealth

25.60 queue

Package queue is a durable, per-destination delivery queue.

Without it, a destination that is down for two minutes loses everything sent during those two minutes: retries happen in memory, and when they run out the message is recorded as failed and gone. With it, the message is on disk and keeps its place until the destination comes back.

Two design decisions shape everything else.

Order is preserved per destination, and that means a queued destination stops taking the fast path. HL7 is a stream of events about the same patients: an A01 admits, an A03 discharges. If a failed A01 goes to the queue while the next A03 is delivered directly, the receiving system sees a discharge for

a patient it never admitted. So once anything is queued for a destination, everything for that destination queues behind it until the queue drains. That costs throughput during an outage and it is not optional, because the alternative is silent clinical nonsense. It is also the specific thing that goes wrong in other engines when somebody turns on concurrent queue threads to make a backlog drain faster.

A queued message is an accepted message. The sender is told AA, because the bytes are on disk and fsynced before the acknowledgement goes out. Saying AE would make a correctly functioning store-and-forward queue look like a fault and invite the sender to resend what we already hold.

Source: github.com/biodream-llc/perfuse/internal/queue

25.61 s3put

Package s3put writes objects to S3 using only the standard library.

The obvious way to talk to S3 is the AWS SDK, and the SDK is enormous. This project sells a static binary with eight direct dependencies and a bill of materials that fits on one screen - that is a real argument with the person who reviews software before a hospital installs it, and trading it away for one connector would be a bad bargain.

What S3 actually needs is an HTTP PUT and a signature. Signature Version 4 is fully specified, it is deterministic, and AWS publishes test vectors for it. So it is implemented here: about two hundred lines against crypto/hmac and net/http, with no new dependency and nothing to patch when the SDK has an advisory.

What this deliberately does not do

No multipart upload, no retries of its own, no credential chain beyond static keys and the standard environment variables. An HL7 message is kilobytes, so multipart solves a problem this does not have; the engine already retries; and an instance-role credential chain would mean IMDS calls, caching and refresh logic, which is where a small implementation stops being small. If somebody needs those, the honest answer is a file destination and a sync tool, not a half-built SDK.

It works with anything speaking the S3 API - MinIO, Ceph, Wasabi, Backblaze - because it is just signed HTTP. That is a side effect worth having, since a hospital object store is often not AWS.

Source: github.com/biodream-llc/perfuse/internal/s3put

25.62 saml

Exclusive XML Canonicalization (exc-c14n) per <https://www.w3.org/TR/xml-exc-c14n/>

This is the byte-exact canonical form that an XML signature signs. Getting it wrong means valid signatures appear invalid and - worse - an implementation that gets it mostly right will pass in testing against its own signatures while rejecting real IdP responses.

The rules that matter for SAML: - Namespace declarations are rendered only if visibly utilised by the element or its attributes. - Inherited namespaces that an element uses but did not declare are rendered

as if declared on that element. - Attributes are sorted: namespace declarations first (sorted by prefix), then other attributes (sorted by expanded name: namespace URI then local name). - Empty elements use start-tag/end-tag, never self-closing. - Text nodes are output verbatim; CDATA is replaced by its content with entity escaping. - Processing instructions and comments outside the document element are excluded.

Source: github.com/biodream-llc/perfuse/internal/saml

25.63 scim

Package scim implements SCIM 2.0, the protocol identity providers use to provision accounts.

Okta, Entra ID, Google Workspace and OneLogin all speak it. An administrator adds somebody to a group in the identity provider, and within a minute an account appears here with the right role. They remove them, and the account goes away.

Deprovisioning is the point

Provisioning is convenience. Deprovisioning is the reason this is worth building, and it is the half that goes wrong.

When somebody leaves an organisation, the identity provider is the system that knows first - usually because HR terminates them and the provider is driven from the HR record. It then tells every connected application to disable the account. If that call fails, or succeeds without actually removing access, a former employee keeps working access to a clinical integration engine and nobody finds out, because nothing is watching for a thing that did not happen.

So the parts of this package that matter most are the ones that make deprovisioning real:

- Setting active to false ends every session immediately, not at the next expiry. A browser tab left open on a laptop that has gone home with a terminated employee must stop working, and a session that survives until it expires is a session that survives the sacking. - A failure is a failure. SCIM has no retry semantics an application can rely on, so returning 200 for something that did not happen means the identity provider records the deprovisioning as complete and never tries again. - Deleting a user is treated the same as disabling: access ends. Audit history is retained regardless, because a record naming somebody who no longer exists is worse than one marked as withdrawn.

Why it is a separate package

SCIM's shapes are not Perfuse's. It has its own error format, its own list envelope, its own patch language, and its own opinions about attribute casing. Keeping the translation in one place means the handlers can be about what happens to an account rather than about JSON.

Source: github.com/biodream-llc/perfuse/internal/scim

25.64 script

Package script runs Mirth Connect JavaScript.

The goal is that a channel's scripts move across untouched. That is a compatibility exercise, not a design exercise, so the shape of everything here is dictated by what Mirth exposes: msg and tmp as E4X trees, six maps with Java's Map methods, the \$ family of lookup functions, logger, and a handful of utility objects whose method names and date-format strings are Java's.

Three decisions are ours rather than Mirth's, and each is a deliberate departure:

A script runs with a deadline and is interrupted when it passes. Mirth will let a script loop forever, which takes the channel with it; an interface that stops accepting admissions because of a typo in a transformer is not acceptable, so there is always a timeout.

Java interoperability is refused with an error naming the class, rather than returning undefined. A script reaching into Packages.java.sql cannot be made to work here, and the useful outcome is a message saying exactly that at the line where it happens, not a NullPointerException three lines later.

File and database access are off unless a channel turns them on. Mirth grants both to every script by default. A transformer that can read any file the process can read is a reasonable thing to want and an unreasonable default.

Source: github.com/biodream-llc/perfuse/internal/script

25.65 settings

Package settings is the single description of everything about a Perfuse server that an operator may change.

Why this exists

Before this package, roughly thirty things were command-line flags and nothing else. Changing how long messages are kept meant editing a service definition and restarting, which in a hospital means a change ticket – so in practice nobody changed them, and a default chosen on the first afternoon became the setting forever.

Why one registry rather than a form

The obvious way to build a settings page is to write the form. That produces two descriptions of every setting – one in the server that validates it and one in the interface that renders it – and they drift. The drift is quiet: a field the server stopped accepting still has a control, so somebody sets it, sees it saved, and it does nothing.

So a setting is declared once, here, with everything needed to both check it and draw it: what kind of value it is, what control suits it, which group it belongs in, what it means, and whether changing it takes

effect now. The interface asks for that description and renders whatever it is told. Adding a setting is one entry in this package and no interface work at all.

TestEverySettingIsFullyDescribed enforces the parts a human has to supply, because a setting with no help text is a setting somebody guesses at.

Why a file rather than the database

Settings live in a YAML file the interface edits in place, matching how channels and alert rules already work. Three reasons, in order of weight: a hospital's change control wants configuration that can be diffed and put in version control; an operator with the machine but not the application can read a file and cannot read a database; and restoring yesterday's settings should not require the application to be working.

Why flags only seed it

The file wins once it exists, and flags supply the initial contents. The alternative - flags always winning - would make the interface a liar, since anything ever passed on a command line could be edited and saved and silently ignored.

A flag that disagrees with the file is reported at startup, naming both values, because otherwise the flag stays in the service definition for years and the next person to read it believes it.

Source: github.com/biodream-llc/perfuse/internal/settings

25.66 sftpconn

Package sftpconn opens SFTP connections.

Split out from the connectors because the interesting part is authentication and host key verification, and both are identical whether a channel is collecting files or writing them. Two copies would drift, and the half that drifted would be the half that stopped verifying.

Source: github.com/biodream-llc/perfuse/internal/sftpconn

25.67 shadow

Package shadow compares a candidate channel against the live one.

The comparison itself is the easy part. The whole value is in two guarantees, and both are enforced by construction rather than by configuration:

- A shadow cannot deliver. It holds a transformation pipeline and no senders, so there is no code path from here to a receiver. A setting cannot create one and a mistake cannot enable one. - A shadow cannot affect the live channel. It is called after the live message has been delivered and acknowledged, its panics are recovered, and its timeouts are its own.

Anything less than that and shadow mode becomes a way to break production while trying to avoid breaking production.

Source: github.com/biodream-llc/perfuse/internal/shadow

25.68 shl

Package shl implements SMART Health Links and SMART Health Cards: the QR codes CMS's "Kill the Clipboard" asks providers to accept at check-in, and to hand back with the visit record.

A SMART Health Link (SMART Health Links STU 1) is shlink:/ followed by base64url JSON: a manifest URL, a 256-bit key, and optionally an expiry, a label and flags (P passcode, L long-term, U a single file by GET). The receiver posts to the manifest URL and gets back files, each a JWE encrypted with that key - alg dir, enc A256GCM, optionally DEFLATE-compressed - holding FHIR JSON, SMART Health Cards, or a SMART access token. The server that hosts the files never sees the key: it travels only in the link, which is the patient's.

A SMART Health Card (SMART Health Cards Framework 1.4) is a signed FHIR bundle: a compact JWS, ES256, with a DEFLATE-compressed payload naming its issuer, whose public key is at <iss>/well-known/jwks.json. It travels as a numeric shc:/ QR code or a .smart-health-card file.

Only the standard library: crypto/aes, crypto/cipher, crypto/ecdsa and compress/flate cover everything, and a JOSE library would bring a much larger surface for two algorithms.

Source: github.com/biodream-llc/perfuse/internal/shl

25.69 smartauth

Package smartauth is a SMART App Launch authorization server: it issues the access tokens Perfuse's FHIR endpoint accepts.

Written against the standard library, like internal/oidc, which verifies what this signs. The tokens are JWTs signed RS256, the one algorithm every SMART client and the specification's ID token rules can rely on.

Source: github.com/biodream-llc/perfuse/internal/smartauth

25.70 soap

Package soap sends SOAP requests and reads SOAP faults, using only the standard library.

SOAP is a fair amount of ceremony around an HTTP POST, and it is still how a great deal of hospital middleware accepts data - patient administration systems, document repositories, regional record services, anything specified in the decade when this was the answer. Mirth has a Web Service connector, so this is a Tier 2 migration blocker rather than a nice-to-have.

Why not a generated client

The usual approach is to take a WSDL and generate Go types from it. That is the right answer for an application talking to one known service, and the wrong one here: an integration engine's destination is configured by somebody editing a channel, not recompiled, so the envelope has to be assembled at run time from a template they can see and change. It also means there is no code generation step and nothing to regenerate when the far end publishes a new WSDL.

What this deliberately does not do

No WSDL parsing, no WS-Security, no MTOM, no WS-Addressing. Those are each a specification in their own right, and pretending to support one while getting a detail wrong is worse than not offering it - a signature that almost verifies is indistinguishable from an attack at the far end. WS-Security in particular is where a half-implementation would do real harm.

What is here: SOAP 1.1 and 1.2 envelopes, a templated body, the SOAPAction header, and fault detection that reads the fault string rather than reporting "500".

Source: github.com/biodream-llc/perfuse/internal/soap

25.71 sqldb

Package sqldb opens database connections for the database source and destination.

The drivers are registered here and nowhere else, so there is exactly one place that decides what Perfuse can connect to.

Source: github.com/biodream-llc/perfuse/internal/sqldb

25.72 sqlitedb

Package sqlitedb opens SQLite the way a server needs it: one connection for writes and several for reads.

The reason is a failure mode that only appears once a deployment has been running for a while. SQLite in WAL mode lets readers run alongside a writer, but Go's connection pool sits above SQLite, so a pool of one serialises everything regardless of what the database would allow. A message browser search scanning the payload column then holds the only connection, and every arriving message waits behind it.

Measured on this machine, a write takes 104µs with the store idle and 2.6ms while a search scans four thousand messages: twenty-five times slower. That scales with the table, so at a few hundred thousand stored messages a report turns into seconds of delivery latency, and a write is on the acknowledgement path. The sender sees a timeout, resends, and somebody spends a morning looking for a problem in the network.

Splitting the pools costs nothing and removes the interaction: reports get their own connections, and the writer keeps one to itself because SQLite permits exactly one writer anyway.

Source: github.com/biodream-llc/perfuse/internal/sqlitedb

25.73 steps

Declarative transformation steps, for any message format.

Why this is shared

The step vocabulary - set, copy, clear, map, replace, trim, case, each with a description and an optional condition - has nothing to do with any particular format. Only three things differ between formats: how a path is written, how a value is read, and how it is written back.

This is the same conclusion the filter grammar reached in `internal/expr`, and for the same reason. X12 had a step engine and NCPDP needed one; copying it would have produced a second four-hundred-line engine to keep in step by hand, and then a third for delimited. The behaviour worth having is subtle enough that a copy would not keep it - the reason the result is re-read after every write, for instance, took a real defect to discover.

What a format supplies

An Accessor: a path compiler, a condition compiler, and an optional veto on steps it cannot honour. Nothing else.

Source: github.com/biodream-llc/perfuse/internal/steps

25.74 stomp

Package `stomp` speaks STOMP 1.2 to a message broker.

This is what "JMS support" has to mean in a program that is not Java. JMS is an API rather than a protocol, so there is nothing to implement: what a site actually needs is for Perfuse to talk to the broker sitting behind their JMS applications.

STOMP rather than OpenWire, and the reasoning is worth recording because the obvious answer is wrong. OpenWire is ActiveMQ's native protocol and a binary serialisation of JMS commands - weeks of work, specific to one broker family, and undocumented except by its implementation. ActiveMQ auto-detects STOMP, AMQP and MQTT on the same port it serves OpenWire on, so a text protocol reaches the same brokers. STOMP covers ActiveMQ Classic, Artemis and RabbitMQ; AMQP 1.0 would add Azure Service Bus and IBM MQ, and is the sensible next one rather than OpenWire.

Written against the standard library. A broker client is not the place for a dependency whose failure modes are somebody else's to explain, and the framing here is small enough that owning it costs less than reading their issue tracker.

Source: github.com/biodream-llc/perfuse/internal/stomp

25.75 store

Package store holds the data that is genuinely stateful: users, sessions and an audit trail.

Channels are deliberately not in here. They live in files, because that is what gives git, diffs, code review and environment promotion for free. Putting channels in a database is the decision that forces every one of those to be bought separately, and it is the reason a third-party product exists purely to shuttle Mirth channels between its database and a git repository.

So the split is: configuration in files, identity and history in SQLite.

Source: github.com/biodream-llc/perfuse/internal/store

25.76 tefca

Package tefca implements the parts of TEFCA participation that do not require a QHIN connection.

What is real

Purpose-of-use validation, configuration validation, and the audit trail - including its persistence, its query interface and its obligation to record failed attempts as well as successful ones. That last part is not a technicality: a run of failed deliveries to one recipient is how a misconfigured partner shows up, and a run of failed queries for different patients is how probing shows up.

What is not

The network. There is no QHIN transport in this build, so no exchange happens, and every exchange function refuses with `ErrExchangeNotImplemented` rather than returning success.

That distinction was not always drawn, and the way it went wrong is worth keeping. All four exchange functions used to validate their inputs, make no network call, and return success; `Deliver` returned an `Accepted` response with a tracking identifier built from the clock. The audit layer then recorded each as a completed exchange - in the log that exists to prove what was exchanged when somebody asks months later. Four tests asserted that behaviour and passed.

What completing it requires

Onboarding with a Qualified Health Information Network, mutually authenticated TLS using certificates issued through that process, and either the IHE profiles the Common Agreement started with or Facilitated FHIR as defined by the Sequoia Project's standard operating procedure, effective 8 March 2026. None of it can be written speculatively: it is tested against a real QHIN or it is not tested, and an untested exchange implementation is the same defect as no implementation wearing a better disguise.

Source: github.com/biodream-llc/perfuse/internal/tefca

25.77 **tenant**

Package `tenant` provides the identity and isolation rules for multi-tenant operation.

An HIE, a clearinghouse or a managed service provider runs one engine on behalf of many organisations. Mirth cannot do this - one instance serves one organisation, and an MSP ends up running fifty JVMs and fifty databases.

What a tenant owns

Its channels, as a directory of files. Its users. Its messages, queue, audit trail and FHIR resources. Its metrics labels. Nothing is shared except the process, the listening ports it was allocated, and the platform administrators.

Isolation is structural, not remembered

The dangerous failure here is not a crash. It is tenant A seeing tenant B's messages, which is a reportable breach rather than a bug. A design where every query takes a `tenant_id` parameter fails the first time somebody writes a query and forgets one, and that query looks perfectly correct in review.

So the tenant is not a parameter. Data access goes through a handle that already carries it, and there is no way to reach the tables without one. Forgetting is not available.

Source: github.com/biodream-llc/perfuse/internal/tenant

25.78 **tlsconf**

Package `tlsconf` builds TLS configuration from files and describes certificates.

This is the free equivalent of an extension Mirth now charges for. Most of the value is not the encryption, which is a few lines, but the part that stops somebody deploying something they think is secure and is not: refusing to skip verification silently, insisting a client certificate is actually checked when one is requested, and saying plainly when a certificate is about to expire.

An expired certificate on an interface feed is a genuine hospital outage, and it happens because nobody was told. The describing half of this package exists for that reason.

Source: github.com/biodream-llc/perfuse/internal/tlsconf

25.79 **tomirth**

Package `tomirth` writes a Perfuse channel back out as Mirth channel XML.

Why this exists. The objection to adopting a new integration engine is rarely "is it any good". It is "what if we are wrong and we are stuck", and the cheapest answer to that is being able to walk back out. Perfuse has read Mirth's channels since the beginning; until now it could not write one, which made the reassurance one-directional and therefore not much of a reassurance.

How it avoids the trap that made the first attempt useless. Mirth's XML is XStream's output, and XStream reads the class and version attributes as instructions: an element it does not recognise is dropped without a word, and a channel it cannot assemble is stored with its description replaced by "This channel is invalid. Verify all required extensions are loaded correctly" and its destinations discarded. Nothing in that response names the element at fault. An exporter that composes property blocks from a reading of the format therefore fails in the one way that is expensive to diagnose, which is what happened: this repository carried an exporter with nine passing tests whose output Mirth threw away, because every test compared it against our own parser instead of a server.

So none of the property blocks are composed here. `scripts/mirth-dump-connector-templates.sh` has Mirth's own `ObjectXMLSerializer` serialise the defaults of each connector properties class, and those documents are committed under `templates/`. This package substitutes values into them, and `mirth.SetRawProperty` refuses any path the template does not already contain - so a mistake is an error naming the path, at the moment of export, rather than a channel that vanishes on import.

What it will not do. A transport with no Mirth equivalent is refused by name rather than approximated. An export that quietly turned an S3 destination into a file writer would be worse than no export at all, because the channel would import cleanly and deliver to the wrong place.

Source: github.com/biodream-llc/perfuse/internal/tomirth

25.80 trace

Package `trace` produces distributed traces for messages moving through a channel, and exports them over OTLP/HTTP with the JSON encoding.

Why this exists rather than the OpenTelemetry SDK: the SDK plus an OTLP exporter pulls in roughly sixty modules and adds about as much to the binary as the whole of the rest of this program. What is actually needed here is narrow - W3C trace context propagation, a span model, head sampling, and a batched HTTP POST - and OTLP/JSON is a documented stable encoding that can be produced with the standard library alone. Same trade as choosing `lib/pq` over `pgx`: take the small thing when only part of the big thing is wanted.

The consequence worth stating: there is no auto-instrumentation and no vendor-specific exporter. Spans are the ones this program creates deliberately, and they go to any OTLP endpoint - a collector, Jaeger, Grafana Tempo, Honeycomb, Datadog's OTLP ingest.

The property that matters more than the spans

A message frequently arrives carrying a `traceparent` header from the system that sent it, and leaves toward a system that will record one of its own. Continuing that trace rather than starting a fresh one is what makes a single view of "the EHR sent this, the engine transformed it, the registry rejected it" possible at all. Interface problems are almost always reported as "we sent it and nothing happened", and that question is unanswerable without an identifier that survives the hop.

Tracing must never affect delivery

Every operation here is best-effort. Spans are dropped when the buffer is full, export failures are counted and never retried into the message path, and no call blocks on the network. A dropped span is an inconvenience. A message delayed or lost because the tracing backend was slow is a clinical incident, so the trade is never in question.

Source: github.com/biodream-llc/perfuse/internal/trace

25.81 transform

Package transform applies declarative changes to a message.

This is the layer meant to carry the ordinary work, with scripting kept as a marked escape hatch. The distinction is not stylistic. A step declared in YAML can be validated before it runs, shown in a diff, reviewed by somebody who does not read JavaScript, and reported on; a script can only be executed and hoped about. Most of what Mirth transformers actually do is boring - copy a field, pad an identifier, map a code through a table, clear something that should not leave the building - and all of that belongs here.

Every step names its target explicitly and every step is reversible in the sense that it says what it did, because the most expensive failure in an interface is a transformation nobody knew was happening.

Source: github.com/biodream-llc/perfuse/internal/transform

25.82 translate

Package translate converts a Mirth channel into a Perfuse channel definition.

`perfuse explain` reports what blocks a migration. This does most of the migration, which is a different and much larger claim, so it is deliberately conservative about what it will assert.

Three rules shape the whole thing.

Nothing is silently dropped. Every part of the source channel is either translated, carried across verbatim as a script, or reported as needing a human. A translator that quietly omitted a step would produce a channel that starts, runs, and loses data — the worst possible outcome and the reason people distrust migration tools.

Declarative where it is certain, script where it is not. A Mapper step that copies a field becomes a declarative step, because that is readable and checkable. A JavaScript step becomes a script, because Perfuse runs Mirth's JavaScript unchanged and rewriting it into something that looks equivalent is exactly where a translator would introduce a difference nobody notices.

The output is meant to be read. It carries comments explaining what came from where, and what was left for a person. A migration is reviewed by somebody who has to sign off on it, and an unannotated wall of YAML cannot be reviewed.

Source: github.com/biodream-llc/perfuse/internal/translate

25.83 udap

Package udap implements the client half of HL7 Security for Scalable Registration, Authentication, and Authorization — the specification TEFCA's Facilitated FHIR exchange rests on.

UDAP is a public key infrastructure bolted onto OAuth 2.0. A trust community issues X.509 certificates to its members; a member proves who it is by signing JWTs with the private key and attaching the certificate chain in the JWT header. That replaces the shared client secret of ordinary OAuth, which does not scale to thousands of organisations who have never met.

Why this package exists at all. TEFCA exchange cannot be fully verified without a real QHIN: joining one needs onboarding and issued certificates, and cannot be faked. But the security profile underneath it is a published HL7 implementation guide with public reference servers, and one of them answers on the open internet. So the part that can be verified is verified here, and the part that cannot is refused at the boundary rather than guessed at.

The distinction matters because of what happened with SAML in this repository: a thousand lines of tests passed, five of them specifically about signature canonicalisation, while no real identity provider could authenticate anybody. Both halves had been written here and agreed with each other. The lesson taken from it is that a security implementation is worth nothing until something somebody else wrote has accepted or rejected it.

Specification references: - HL7 UDAP Security STU 1: <http://hl7.org/fhir/us/udap-security/STU1/> - TEFCA SOP Facilitated FHIR Implementation v2.0, effective 8 March 2026

Source: github.com/biodream-llc/perfuse/internal/udap

25.84 v2fhir

Package v2fhir maps HL7 v2 messages to FHIR resources.

Mapping is where interoperability projects actually fail, so the rules this package follows are stated rather than implied:

- A value that cannot be mapped is preserved as text, never guessed at. A local code turned into a plausible-looking standard code is worse than an uncoded value, because the receiver believes it. - Every mapping decision that involved judgement is reported as a note, so a human can check the ones that matter instead of reading the whole output. - Timestamps are converted, not copied. HL7 v2 timestamps are not ISO 8601, and a dateTime carrying a local time with no offset is the single most common reason a converted resource is rejected. - Units keep their original text and gain a UCUM code only when the mapping is certain. A wrong unit code is a patient-safety defect, not a formatting problem.

Source: github.com/biodream-llc/perfuse/internal/v2fhir

25.85 vcs

Package vcs reads the history of channel files from git.

Channel history is a paid extension elsewhere, and it is a paid extension because those engines keep channels in a database, so tracking changes means building a change-tracking system. Perfuse keeps channels in files, so the history already exists and has since the first commit. This package only has to read it.

It shells out to git rather than embedding a git library. The library would add a large dependency to a binary whose selling point is having four, and it would buy nothing: a deployment keeping its channels under version control has git installed, and one that does not gets a clear "not a git repository" rather than a subtly different implementation of the same thing.

Source: github.com/biodream-llc/perfuse/internal/vcs

25.86 vfs

Package vfs is the small set of file operations a polling connector needs.

Why an interface rather than four connectors

Perfuse collects files from a local disk, an SFTP server, an FTP server, a Windows share and a WebDAV server. The transfer differs in every case. Everything that actually decides whether messages survive does not:

- A file being written and a file finished being written are indistinguishable. There is no lock, no flag and no notification on any of these transports - there is a size and a modification time, and both are true of a half-written file. Reading too early collects half a message, and HL7 has no terminator, so half a message is very often still parseable. The MSH is intact, the segments that arrived are well formed, the ones that did not are simply absent. It is accepted, acknowledged, stored, delivered, and nothing will ever mention it. - A file must not be disposed of until every message in it has been accepted, because the file is the only copy. - A file that failed must not be picked up again on the next poll forever, and must not be silently dropped.

Written once per transport, those rules would be subtly different five times, and the differences would only appear as messages that went missing at one site. So they are written once, here, over the smallest interface that supports them.

What is deliberately not here

No writing, beyond what disposal needs. Sending files is a different problem with different failure modes - a partially written file at the far end, which the existing senders handle by writing to a temporary name and renaming. Folding both directions into one interface would make each harder to reason about.

No recursion. A poller that descends into subdirectories collects the archive directory it just moved a file into.

Source: github.com/biodream-llc/perfuse/internal/vfs

25.87 **vpn**

Package `vpn` reports the state of the site-to-site VPN tunnels partner traffic runs over - AWS Site-to-Site VPN, Azure VPN Gateway connections and strongSwan - and keeps each partner's connection sheet: both sides' tunnel settings, and whether they agree.

A tunnel that is down looks, from an interface engine, exactly like a partner whose engine is stopped: connections time out. Saying "the tunnel is down" instead is the difference between calling the network team and calling the partner.

Read only. Nothing here changes a tunnel, and no pre-shared key is ever read, kept or shown.

Source: github.com/biodream-llc/perfuse/internal/vpn

25.88 **web**

Package `web` serves the built front end from inside the binary.

The assets are embedded rather than shipped alongside, so deployment stays one file. That is the same reason the engine avoids `cgo`: an operator should be able to copy a binary onto a server and run it, with no JVM, no installer and no asset directory to get out of step with the executable.

The built output is committed so that `go build` works on a checkout without Node installed. Rebuild it with `make web`.

Source: github.com/biodream-llc/perfuse/internal/web

25.89 **webauthn**

Package `webauthn` verifies passkeys.

A passkey is a key pair the authenticator holds - Touch ID, Windows Hello, a YubiKey, a phone - where the private half never leaves the device. Signing in means the server sends a random challenge, the authenticator signs it after checking the user is present, and the server verifies the signature against the public key it stored at registration.

The property that matters is not that it is convenient. It is that there is nothing to phish. A password can be typed into a convincing copy of a login page; a passkey signature is bound to the origin it was created for, so the copy gets a signature its server cannot use. For an integration engine holding a hospital's interfaces, that is the difference worth having.

What this implementation refuses to do

The specification is permissive in places where being permissive is how implementations get broken. This one is not:

- The signing algorithm is fixed at registration and stored with the key. Nothing at sign-in reads an algorithm from the client, because that is algorithm confusion and there is no function here that

could do it. - The origin is compared exactly, never by prefix or suffix. A check that accepts anything ending in the right domain accepts an attacker's subdomain. - A challenge is single-use and expires. It is deleted when consumed, so a captured assertion cannot be replayed. - Attestation is not trusted for anything. The attestation statement is parsed to extract the credential and then its signature is ignored, because verifying it properly needs a maintained root store and pretending to verify it is worse than not claiming to. - The sign count is checked when the authenticator provides one, and its absence is not treated as suspicious. Many authenticators legitimately leave it at zero.

Source: github.com/biodream-llc/perfuse/internal/webauthn

25.90 winservice

Package winservice runs Perfuse as a Windows service.

This file is the everywhere-else half. It exists so that cmd/perfuse can call into this package without wrapping every call site in a build tag: the subcommand is always compiled, and on Linux or macOS it explains that services are a Windows idea rather than failing to exist.

Refusing with a clear sentence is better than hiding the subcommand. Somebody following Windows instructions on the wrong machine gets told what is actually going on, and the equivalent for their platform.

Source: github.com/biodream-llc/perfuse/internal/winservice

25.91 x12

Reading and writing values at a path.

Why this is separate from Path

path.go decides what a path expression means. This file resolves one against an actual interchange. Kept apart because the parsing rules are worth reading without the byte handling, and because the parser is used by the configuration validator, which has no message to resolve against.

Why a write re-parses instead of editing bytes in place

Segment.elements are subslices of Segment.raw, and Message.segs indexes into Message.raw. Editing a value in place changes the length of everything after it, so every one of those slices would need adjusting, and any that was missed would read correct until the one message where it did not.

So a write rebuilds the segment, rebuilds the interchange, and parses it again. That costs a parse per changed message and buys the guarantee that a Message returned from here is exactly what parsing its own bytes produces. There is no state where the indexes and the bytes disagree, because that state is unrepresentable.

Source: github.com/biodream-llc/perfuse/internal/x12

25.92 x12xml

Package x12xml converts an X12 interchange to and from the node tree the transformation, filter and script layers already work on.

This is the same trick hl7xml plays, and for the same reason. Everything above the parser - declarative transformation steps, the filter language, JavaScript, the channel test runner - operates on an `xtree.Node` rather than on a format. Render X12 into that tree and all of it works on claims without a line of new code in any of those packages. Write a second addressing layer instead and every one of them grows an X12 branch that has to be kept in step with the HL7 one for ever.

Element naming

Elements are named in implementation-guide form: ISA13, CLM01, NM103.

Two independent reasons point the same way. It is what every 837 and 835 implementation guide writes, so a person reading the guide and a person reading the tree see the same token. And it is what Mirth's own X12 serialiser produces, so a Mirth script that walks `msg['CLM']['CLM01']` keeps working - which is the whole migration promise.

The element number is always two digits, matching the guides, so CLM01 rather than CLM1. Composite components add a further dotted number, CLM05.1, because X12 composites are one level deep and there is nothing to collide with.

Source: github.com/biodream-llc/perfuse/internal/x12xml

25.93 xmldsig

Package xmldsig implements XML Digital Signatures and the XAdES profile of them.

Why this is written rather than borrowed

A signed clinical document is a legal artefact. It asserts that a named person, at a named time, took responsibility for a specific set of bytes. Getting that wrong in either direction is serious: a signature that cannot be verified makes a valid document look forged, and a verifier that is too generous accepts a document somebody altered.

Canonicalisation is the whole problem

XML has many byte sequences that mean the same thing. Attribute order is not significant, namespace prefixes can be renamed, empty elements can be written two ways, and whitespace inside a tag is free. A signature is over bytes, so before hashing anything both ends must agree on exactly one byte sequence for a given document. That agreement is canonicalisation, and it is where implementations go

wrong, because every mistake produces a signature that verifies against your own output and fails against everybody else's.

So this file is the canonicalisation, on its own, with its own tests. Nothing about signing appears here.

Exclusive rather than inclusive, by default

Inclusive canonicalisation pulls in every namespace declaration that is in scope, including ones the signed element never uses. That makes a signature break when the document is later embedded inside something that declares an unrelated namespace on an ancestor, which is precisely what happens when a CDA is put inside a SOAP envelope or an IHE metadata wrapper. Exclusive canonicalisation includes only the declarations the element actually uses, and is what document exchange profiles specify for that reason.

Source: github.com/biodream-llc/perfuse/internal/xmlsig

25.94 xtree

Package xtree is a mutable XML tree.

Three very different features need to modify XML: HL7 transformations, which Mirth models as XML because that is what its scripts operate on; clinical documents, which are XML natively; and the declarative transformation steps. They share this one tree rather than each growing their own, because the alternative is three models that disagree about what an empty element means.

The design goal is faithfulness to E4X semantics, since Mirth scripts are E4X and the whole point of running them unchanged is that they behave the same. That drives two decisions that would otherwise look strange: a node keeps a pointer to its parent, because E4X exposes `parent()`, and children are ordered with duplicates allowed, because an HL7 field repetition is literally a repeated element and scripts index into them by position.

Source: github.com/biodream-llc/perfuse/internal/xtree

26 Index of Configuration Keys

Every key in alphabetical order, with the blocks it can appear in. A key that appears in several blocks means the same thing in each one; where it does not, the blocks are documented separately and the difference is stated there.

26.1 All keys

Key	Appears in
access_key_id	AWS Access, FHIR Auth, S3 Destination
account	Azure Blob Access
ack	HTTP Source, SOAP Source, Source
ack_sender_id	X12 Options
ack_sender_qualifier	X12 Options
acknowledge	HL7v3 Options, X12 Options
acks	Kafka Destination
action	SOAP Destination
addr	AMQP Connection, Broker Destination, Broker Source, MQTT Connection
address	AMQP Connection, DICOM Destination, DICOM Query Source, Destination, TCP Dest
ae_title	DICOM Source
after_query	Database Source
after_read	Azure Blob Source, File Poll, GCS Source, S3 Source, SFTP Source
allow	Scripts
allow_clear_password	FTP Destination
allowed_calling_ae	DICOM Source
amqp	Destination, Source
api_version	IBMMQ Connection
append	SFTP Destination
application	Ack

Key	Appears in
attach	SMTP Destination
attach_name	SMTP Destination
attachments	Channel
attempts	Retry
auth	FHIR Destination
azure_blob	Destination, Source
backoff	Queue Config, Retry
base64	SOAP Source
batch_size	Database Source, File Poll
baud	Serial Source
bcc	SMTP Destination
bearer_token	CDA Destination, FHIR Destination, HTTP Destination
big_endian	TCP Framing
blob	Azure Blob Destination
body	SMTP Destination, SOAP Destination
broker	Destination, Source
brokers	Kafka Destination, Kafka Source
bucket	GCS Access, S3 Destination, S3 Source
called_ae	DICOM Destination, DICOM Query Source
calling_ae	DICOM Destination, DICOM Query Source
cc	SMTP Destination
cda	Destination
channel	Destination, Shadow

Key	Appears in
check_every	Contract Ref
city	ECR Facility
claim_us_core	FHIR Destination
client_id	FHIR Auth, MQTT Connection
client_secret	FHIR Auth
column	Database Source
columns	Delimited
comment	Delimited
commit_after_delivery	Kafka Source
compare	Shadow
compression	Kafka Destination
config	ELR Destination
container	Azure Blob Access
content_type	AMQP Destination, Azure Blob Destination, Broker Destination, GCS Destination, HTTP Destination, S3 Destination
contract	Channel
credentials_file	GCP Access
dataType	Channel
data_bits	Serial Source
database	Destination, Source
default_identifier_system	FHIR Destination
delimited	Channel
delimiter	Delimited, TCP Framing

Key	Appears in
deploy	Scripts
description	Channel
destination	Broker Destination, Broker Source
destinations	Channel
dicom	Channel, Destination, Source
dicom_query	Source
dir	CDA Destination, Destination, Document Destination, FTP Destination, File Poll, SFTP Destination, SFTP Source
document	Destination
domain	SMB Source
driver	Database Destination, Database Source
dsn	Database Destination, Database Source
ecr	FHIR Destination
element	SOAP Source
elr	Destination
emit_on_first_poll	DICOM Query Source
enabled	Channel, Destination, Queue Config
endpoint	AWS Access, Azure Blob Access, GCP Access, S3 Destination
envelope	X12 Options
error_dir	File Poll, SFTP Source
error_prefix	Azure Blob Source, GCS Source, S3 Source
expect_reply	TCP Dest
extract	Attachments

Key	Appears in
facility	Ack, ECR Destination
fail_on_body	HTTP Destination
fault_is_success	SOAP Destination
fault_on_nak	SOAP Source
fhir	Destination
file	Contract Ref, Source
file_name	Document Destination, FTP Destination, SFTP Destination
file_roots	Scripts
filter	Channel, Delimited, Destination, HL7v3 Options, Scripts
flow_control	Serial Source
follow_redirects	HTTP Destination
follow_symlinks	File Source
font_size	Document Destination
format	Document Destination, S3 Destination
framed	Azure Blob Source, FTP Destination, File Poll, GCS Source, S3 Destination, S3 Source, SFTP Destination, SFTP Source
framing	TCP Framing
from	SMTP Destination
from_beginning	Kafka Source
ftp	Destination, Source
gcs	Destination, Source
group	Channel, Kafka Source
group_by	SNS Destination, SQS Destination

Key	Appears in
has_header	Delimited
header	SOAP Destination
headers	Broker Destination, CDA Destination, FHIR Destination, HTTP Destination, Kafka Destination, SOAP Destination
heartbeat	Broker Source
hl7v3	Channel
host	Broker Destination, Broker Source, FTP Destination, FTP Source, SFTP Destination, SFTP Source, SMB Source, SMTP Destination
hostname	AMQP Connection
http	Destination, Source
ibm_mq	Destination, Source
identifier_systems	CDA Destination, FHIR Destination
idle_timeout	DICOM Source, Source, TCP Source
ignore	Shadow
include	Scripts
include_trigger_event	Ack
insecure_skip_host_key_check	SFTP Destination, SFTP Source
insecure_skip_verify	FTP Destination, FTP Source, SMTP Destination, Web DAV Source
interval	DICOM Query Source
javascript	Destination, Source
kafka	Destination, Source
keep_alive	TCP Dest
keep_blank_lines	Delimited
keep_delimiter	TCP Framing

Key	Appears in
key	Azure Blob Access, Kafka Destination, S3 Destination
key_column	Database Source
key_file	SFTP Destination, SFTP Source
key_passphrase	SFTP Destination, SFTP Source
known_hosts_file	SFTP Destination, SFTP Source
landscape	Document Destination
language	Scripts
length_bytes	TCP Framing
length_includes_header	TCP Framing
level	DICOM Query Source
limit	DICOM Query Source
line	ECR Facility
listen	DICOM Source, HTTP Source, SOAP Source, Source, TCP Source
login	Broker Destination, Broker Source
match	DICOM Query Source
max_attempts	Database Source, Queue Config
max_backoff	Queue Config, Retry
max_connections	Source, TCP Source
max_depth	Queue Config
max_differences	Shadow
max_file_size	File Poll, SFTP Source
max_message_size	HTTP Source, Kafka Source, SOAP Source, Serial Source, Source, TCP Dest, TCP Source

Key	Appears in
max_messages	Pub Sub Source, SQS Source
max_object_bytes	DICOM Source
max_object_size	Azure Blob Source, GCS Source, S3 Source
max_open_conns	Database Destination
mechanism	Kafka SASL
method	HTTP Destination
move_to	Azure Blob Source, File Poll, GCS Source, S3 Source, SFTP Source
mqtt	Destination, Source
name	Channel, Channel Destination, Destination, ECR Facility
ncpdp	Channel
non_persistent	IBMMQ Destination
npi	ECR Facility
object	GCS Destination
on_no_document	CDA Destination
order_by	Pub Sub Destination
over	Contract Ref
overlap	DICOM Query Source
params	Database Destination
parity	Serial Source
passcode	Broker Destination, Broker Source
password	AMQP Connection, FTP Destination, FTP Source, HTTP Destination, IBMMQ Connection, Kafka SASL, MQTT Connection, SFTP Destination, SFTP Source, SMB Source, SMTP Destination, SOAP Destination, SOAP Source, Web DAV Source

Key	Appears in
path	HTTP Source, SOAP Source
path_style	S3 Destination, S3 Source
patient_root	DICOM Query Source
pattern	File Poll, SFTP Source
persistent	Broker Destination
phone	ECR Facility
poll_interval	Azure Blob Source, Database Source, File Poll, GCS Source, IBMMQ Source, Java Script Source, S3 Source, SFTP Source
port	Serial Source
postal_code	ECR Facility
postprocessor	Scripts
prefetch	AMQP Source
prefix	Azure Blob Source, GCS Source, S3 Source
preprocessor	Scripts
project	GCP Access
pubsub	Destination, Source
qos	MQTT Connection
query	Database Source
query_timeout	Database Source
queue	Destination, IBMMQ Connection
queue_manager	IBMMQ Connection
queue_url	SQS Destination, SQS Source
quiet_after	Serial Source

Key	Appears in
quote	Delimited
raw	File Poll
rctc	ECR Destination, ELR Destination
read_timeout	HTTP Source, SOAP Source
reassemble	Attachments
reconnect	AMQP Source, Broker Source, MQTT Source, Pub Sub Source
record_length	TCP Framing
region	AWS Access, FHIR Auth, S3 Destination
reject_on_warning	FHIR Destination
relaxed	Delimited
reopen_after	Serial Source
reply	Serial Source, TCP Source
reply_text	Serial Source, TCP Source
require_agreement	CDA Destination
response_element	SOAP Source
response_namespace	SOAP Source
response_transformer	Destination
retain	MQTT Destination
retain_hours	Queue Config
retry	Destination
return	DICOM Query Source
root	FTP Source, File Source, SMB Source
s3	Destination, Source

Key	Appears in
sample	Shadow
sas	Azure Blob Access
sasl	Kafka Destination, Kafka Source
scope	FHIR Auth
script	Channel, Java Script Destination, Java Script Source
scripts	Channel
secret_access_key	AWS Access, FHIR Auth, S3 Destination
security	FTP Destination, FTP Source
selector	Broker Source
sender_device	HL7v3 Options
sender_oid	HL7v3 Options
serial	Source
server_side_encryption	S3 Destination
session_timeout	Kafka Source
session_token	AWS Access, FHIR Auth, S3 Destination
sftp	Destination, Source
shadow	Channel
share	SMB Source
smb	Source
smtp	Destination
sns	Destination
soap	Destination, Source
sop_classes	DICOM Source

Key	Appears in
sort_by	File Poll
source	Channel, ECR Destination
split	Delimited, X12 Options
sqs	Destination, Source
stable_for	File Poll, SFTP Source
start_block	TCP Framing
starttls	SMTP Destination
state	ECR Facility
statement	Database Destination
stop_bits	Serial Source
storage_class	GCS Destination, S3 Destination
subject	SMTP Destination, SNS Destination
subscription	Pub Sub Source
subscription_id	Broker Source
success_on_undefined	Java Script Destination
success_status	HTTP Destination
suffix	Azure Blob Source, GCS Source, S3 Source
tables	Channel
tcp	Destination, Source
temp_suffix	Document Destination, FTP Destination, SFTP Destination
template	Database Source, Document Destination
tenant_id	FHIR Auth
tier	Azure Blob Destination

Key	Appears in
timeout	AMQP Connection, Broker Destination, Broker Source, DICOM Destination, DICOM Query Source, Database Destination, Destination, Document Destination, FTP Destination, FTP Source, IBMMQ Connection, Java Script Destination, Java Script Source, Kafka Destination, Kafka Source, MQTT Connection, S3 Destination, SFTP Destination, SFTP Source, SMB Source, SMTP Destination, SOAP Destination, Scripts, Shadow, TCP Dest, Web DAV Source
timezone	FHIR Destination
title	Document Destination
tls	AMQP Connection, Broker Destination, Broker Source, DICOM Destination, DICOM Query Source, DICOM Source, Destination, HTTP Destination, HTTP Source, IBMMQ Connection, Kafka Destination, Kafka Source, MQTT Connection, SOAP Destination, SOAP Source, Source, TCP Dest, TCP Source
to	SMTP Destination
token	HTTP Source, SOAP Source
token_url	FHIR Auth
topic	Kafka Destination, MQTT Destination, MQTT Source, Pub Sub Destination
topic_arn	SNS Destination
topics	Kafka Source
transfer_syntaxes	DICOM Destination, DICOM Source
transformations	Channel, DICOM Options, Delimited, HL7v3 Options, NCPDP Options, Script Options, X12 Options
transformer	Scripts
trim_padding	TCP Framing
trim_space	Delimited
type	Destination, FHIR Auth, Source

Key	Appears in
undeploy	Scripts
url	CDA Destination, FHIR Destination, HTTP Destination, IBMMQ Connection, SOAP Destination, Web DAV Source
user	FTP Destination, FTP Source, SFTP Destination, SFTP Source, SMB Source, Web DAV Source
username	AMQP Connection, HTTP Destination, IBMMQ Connection, Kafka SASL, MQTT Connection, SMTP Destination, SOAP Destination, SOAP Source
validate_before_send	FHIR Destination
version	CDA Destination, FHIR Destination, SOAP Destination, SOAP Source
visibility_timeout	SQS Source
wait_seconds	SQS Source
webdav	Source
when	Ack
window	DICOM Query Source
write	CDA Destination
wsdl	SOAP Source
x12	Channel