RedRuff Solve

Programming Language

Official Developer Guide

Beginner friendly. Production ready. Built for real-time solutions.

Built by **Sai Pavan Velidandla**

Founder & CEO, RedRuff

pavan@redruff.in +91 9502 444 362

https://redruff.in https://solve.redruff.in

Version 7.1.0. This guide teaches the Solve language and runtime concepts.

Contents

How to Use This Guide

This guide is written for people who are new to Solve, including students who have just started programming. It does not assume that you already know databases, queues, rules engines, workflow engines, or observability systems.

The goal is simple: by the end, Solve should feel easy. If learning the language from raw syntax feels like a 4 out of 10, learning it through this guide should feel like a 1 out of 10.

Read it in order if you are new. Experienced developers can jump to chapters on events, plans, connectors, traces, or operations.

- Part I explains why Solve exists.
- Part II teaches the language from zero.
- Part III teaches real-time behavior: events, reactions, schedules, plans, human steps, connectors, and traces.
- Part IV builds real-world solutions through stories, algorithms, and code mapping.
- Part V covers operations, performance, testing, and when not to use Solve.

Note. This guide does not deeply cover Studio. Studio is an interface for working with Solve, but the language and runtime are the focus here. A separate Studio handbook can cover visual workflows, screens, and daily product usage.

Part I

Why Solve Exists

1 The Problem Solve Was Built For

1.1 The real problem

Most software that runs a business is not just about showing screens or storing data. It has to answer questions like:

- Is this order valid?
- What should happen when stock becomes low?
- Which driver should get this delivery?
- Can we approve this vendor with the information we have?
- What should happen if a human reviewer rejects a step?
- Why did the system choose this action?
- Can we recover safely after a crash?

Traditional programming languages can solve these problems, but usually by combining many separate systems: a database, background jobs, cron, workflow orchestration, rule checks, API clients, audit logging, queues, dashboards, and custom glue code. The hard part is not writing one function. The hard part is making all parts agree about state, truth, timing, failure, retries, and explanation.

Solve was built to make those ideas native.

1.2 Why another programming language?

There are already many excellent languages. Python is approachable. JavaScript is everywhere. Java, C#, Go, and Rust are powerful for services. SQL is excellent for data. Workflow engines are good at orchestration. Rules engines are good at policy. Queues are good at background work.

The issue is that real-time operational software often needs all of them at once.

Need	Typical stack	Solve approach
Persistent state	Database + ORM	Entities and SolveDB
Rules that must remain true	App code + constraints	Rules and transactions
React to changes	Queue + worker	Reactions
External events	Webhooks + queue	Events and handlers
Scheduled checks	Cron + worker	Schedules
Best valid choice	Custom scoring code	Problems and decisions
Human workflows	Workflow engine	Plans and human steps
Auditability	Logs + tracing	Reasoning traces
Connect to systems	API clients	Configurable connectors
Recovery	Custom operations	Built-in backup, restore, recovery

Solve is not trying to replace every language for every job. It is designed for a specific category: real-time, stateful, rule-driven, explainable operational systems.

1.3 The core promise

> **Model the state. Preserve what must be true. React to change. Select the best valid outcome. Explain everything.**

That sentence is the entire language philosophy.

1.4 A tiny example

```
entity Task:
    title: Text
    done: Bool = false

when Task.done becomes true:
    show "Completed: \(Task.title)"
```

This small program already shows the style:

- `entity Task` models real data.
- `done: Bool = false` gives a default.
- `when Task.done becomes true` reacts to a change.
- `show` explains that something happened.

Try it. Write an entity named `Reminder` with fields `text: Text` and `sent: Bool = false`. Add a reaction that prints `"Reminder sent"` when `sent` becomes true.

2 Where Solve Fits in the Market

2.1 Solve compared with common approaches

The comparison below is not about declaring one tool universally better. It is about showing which problems each category is built for.

Category	Strength	Common gap in real-time operations	Solve position
Python scripts	Very easy to start, huge ecosystem	You still assemble database, jobs, rules, traces, queues	Solve is more domain-native for state + rules + reaction + trace
Web frameworks	Great for APIs and apps	Business behavior spreads across controllers, jobs, database code	Solve keeps domain behavior close to the model
SQL databases	Excellent storage and querying	Do not describe workflows, decisions, traces, human steps	SolveDB is integrated with language behavior
Job queues	Reliable background work	Do not know domain truth or why a job exists	Reactions, events, and schedules are domain-aware
Workflow engines	Strong orchestration	Often separate from source-level domain model	Plans and steps live inside the same language
Rules engines	Good rule evaluation	Usually need separate state, workflow, and trace integration	Rules are part of transactions and traces
Observability tools	Great monitoring	Observe after the fact; not usually part of language semantics	Reasoning traces are produced by solve/decide/-plan behavior

2.2 What makes Solve novel

Solve combines several ideas that are usually separate:

1. A readable programming language.
2. An embedded database model.
3. Rules and transactions.
4. Reactions to state changes.
5. Events and schedules.

6. Declarative problem solving and decisions.
7. Durable plans, steps, human steps, and compensation.
8. Configurable connectors and secrets.
9. Audit-grade traces.
10. Operational recovery commands.

The novelty is not one single keyword. The novelty is the unified mental model.

2.3 Measured release evidence

RedRuff Solve Enterprise 7.1.0 was validated in a Windows x86-64 release environment. Exact performance depends on hardware and workload, but these release-gate measurements show the system is lightweight and fast for the validated scope.

Measurement	7.1.0 release-gate result
Parser throughput	about 53.6M characters/sec
Easy syntax lowering	about 5.46M/sec
SolveDB writes	about 226k records/sec
SolveDB reads	about 2.78M records/sec
Reaction matching	about 1.27M/sec
Plan execution	about 248/sec
Scheduler throughput	about 340/sec
Trace journal writes	about 641/sec
One-million-record load writes	about 270k/sec
One-million-record paginated reads	about 3.46M/sec
Startup p95	about 20.5 ms
Studio API p95	about 25.4 ms
One-million-load peak memory	about 3.08 GB under 4 GB threshold

Note. These numbers are included to give realistic expectations, not to promise identical results on every machine. Benchmarking should always be repeated with your own workload.

2.4 When Solve is the right choice

Use Solve when the domain has changing state, rules, reactions, choices, deadlines, human steps, connectors, and audit needs.

Examples:

- Logistics dispatch
- Inventory replenishment
- Vendor onboarding
- Incident response
- SLA monitoring
- Customer support escalation

- Loan or application approval
- Compliance review
- Scheduling and operations control

2.5 When Solve may not be the right choice

Use a traditional stack when the application is mostly:

- A static website
- A graphics-heavy game
- A low-level device driver
- A pure numerical simulation requiring specialized libraries
- A simple script with no state, rules, or audit trail
- A massive distributed database problem where distribution is the primary challenge

Solve is strongest when the central question is: *what should happen next, and why?*

Part II

Learning the Language from Zero

3 The Basic Shape of Solve Code

3.1 Indentation and blocks

Solve uses indentation to show structure. A colon starts a block.

```
function greet(name: Text) -> Text:
    return "Hello, \(name)"
```

You can read this as: define a function named `greet`; it takes a `Text`; it returns `Text`; its body is indented.

3.2 Names and values

A value can be stored in a name:

```
name = "Asha"
age = 21
active = true
```

Common basic types:

Type	Meaning
`Text`	text such as names, emails, notes
`Int`	whole numbers
`Float`	decimal numbers
`Decimal`	money-like or precise decimal values
`Bool`	`true` or `false`
`Instant`	point in time
`Duration`	amount of time
`UUID`	unique identifier

3.3 Functions

Functions are reusable calculations or actions.

```
function addTax(amount: Money, taxRate: Decimal) -> Money:
    return amount + (amount * taxRate)
```

3.4 Conditions

```
if order.total > 10000:
    show "High value order"
else:
    show "Normal order"
```

3.5 Loops

```
for item in order.items:
    show item.name

while retryCount < 3:
    retryCount += 1
```

Try it. Write a function named `isHighPriority` that accepts `score: Int` and returns `true` when the score is greater than or equal to 80.

4 Entities: Modeling the Real World

4.1 What is an entity?

An **entity** is persistent business state. If your application needs to remember it, it is probably an entity.

Examples:

- Customer
- Order
- Product
- Driver
- Vendor
- Incident
- Task

```
entity Customer:
    id: UUID
    name: Text
    email: Text
    active: Bool = true
```

4.2 Default values

A field can have a default value:

```
entity Order:
    status: Text = "pending"
    total: Money
    createdAt: Instant = Clock.now()
```

4.3 Creating records

Use `create Type:` to create persistent data.

```
customer = create Customer:
    name = "Asha Rao"
    email = "asha@example.com"
```

4.4 Updating records

Updates should happen inside a `transaction` when they must be saved safely.

```
transaction:
    order.status = "paid"
    save order
```

4.5 Structures versus entities

A **structure** is a typed value. It is useful for API payloads, function results, connector inputs, or temporary data.

```
structure Address:
    line1: Text
    city: Text
    postalCode: Text

structure InventoryCheck:
    available: Bool
    remaining: Int
```

Use an entity when the system must remember and manage the object. Use a structure when you need a typed value.

Try it. Create an entity `Book` with `title`, `author`, `copiesAvailable`, and `createdAt`. Create a structure `BorrowResult` with `approved` and `reason`.

5 Queries: Asking Questions About Data

5.1 Readable query syntax

Queries read like sentences.

```
lowStock = Product where inventory < 10 sort by inventory limit 50
```

Read it as: from Product, keep rows where inventory is less than 10, sort by inventory, and keep at most 50.

5.2 Filtering

```
pendingOrders = Order where status == "pending"
largeOrders = Order where total > 10000
southDrivers = Driver where region == "south" and status == "available"
```

5.3 Sorting and limiting

```
recentOrders = Order where status == "pending" sort by createdAt limit 100
```

5.4 Why queries matter in real time

Most decisions begin by finding the relevant records:

- available drivers
- low-stock products
- open incidents
- overdue tasks
- vendors missing documents

5.5 Query pattern

Pattern. Use queries to form the candidate set. Then use rules, requirements, decisions, or problems to choose what should happen.

```
candidates = Driver where status == "available" and region == order.region
```

Try it. Write a query that finds active customers whose risk score is above 70, sorted by risk score, limited to 20.

6 Transactions: Changing State Safely

6.1 Why transactions exist

A transaction groups changes so they succeed or fail together. This matters when changing business state.

Story. A customer buys the last item in stock. If the order is saved but inventory is not reduced, the business may sell the same item twice. If inventory is reduced but the order fails, the stock count becomes wrong. A transaction keeps the change together.

6.2 Basic transaction

```
transaction:
    require product.inventory >= order.quantity
    product.inventory -= order.quantity
    save product
    save order
```

6.3 Requirements inside transactions

A `require` is a condition that must be true. If it is false, the transaction does not continue.

```
transaction:
    require account.balance >= payment.amount
    account.balance -= payment.amount
    save account
```

6.4 Algorithm-to-code mapping

Algorithm step	Solve mapping
Check whether stock exists	`require product.inventory >= order.quantity`
Reduce inventory	`product.inventory -= order.quantity`
Persist product	`save product`
Persist order	`save order`
Fail safely if any step is invalid	transaction semantics

Try it. Write a transaction that transfers reward points from one customer to another. Prevent negative balances.

7 Rules and Policies: Preserving Truth

7.1 Rules

A **rule** describes something that must remain true.

```
rule InventoryCannotBeNegative:
    always Product.inventory >= 0
```

A rule is not just a comment. It protects the model.

7.2 Rules versus requirements

Use a rule when the truth should apply globally. Use `require` when the truth applies to a specific operation.

Construct	Use when
`rule`	The condition is always true for the domain
`require`	The condition is needed in this function, transaction, step, or problem

7.3 Policies

Policies protect who may read or change data.

```
policy OrderAccess for Order:
    allow read when user.role == "support"
    allow update when user.role == "dispatcher"
    deny delete
```

7.4 Capabilities

Capabilities protect external effects. A connector, file operation, network call, or secret should not be available unless the project grants it.

```
policy ConnectorAccess:
    allow call Connector.mail.sendEmail when user.role == "support"
```

Try it. Add a rule that an order total can never be negative. Add a policy that only users with role `"manager"` can approve refunds.

8 Reactions: Responding to State Changes

8.1 What is a reaction?

A **reaction** runs when stored state changes.

```
when Order.status becomes "paid":
    show "Order paid: \(Order.id)"
```

8.2 Why reactions are useful

Reactions express domain consequences:

- When stock is low, create a purchase request.
- When an order is delayed, notify support.
- When a vendor is approved, activate the account.
- When an incident becomes critical, escalate.

8.3 A practical reaction

```
when Product.inventory becomes 0:
    create StockAlert:
        product = Product
        message = "Product is out of stock"
        priority = "high"
```

8.4 Reactions and reliability

In production, reactions must not be forgotten after a crash. In Solve 7.1, service-mode reaction dispatch is supervised, bounded, retryable, and traceable in the validated release scope.

Try it. Create a reaction that opens a support ticket when an order status becomes `"failed"`.

Part III

Solving and Decisions

9 Problems: Selecting the Best Valid Outcome

9.1 What is a problem?

A **problem** chooses the best valid candidate from a set. It is not a black box. It filters invalid candidates, scores valid ones, and explains the result.

Story. A delivery order arrives. Five drivers are nearby, but only three are available. One cannot carry the weight. Two remain. The system should choose the best driver and explain why.

9.2 Basic problem

```
problem AssignDriver(order: Order) -> Driver:
    choose driver from Driver where status == "available"

    require driver.region == order.region
    require driver.capacity >= order.weight

    minimize driver.currentLoad weight 0.6
    maximize driver.rating weight 0.4

    explain
```

9.3 How to read it

Line	Meaning
`choose driver from Driver`	Candidate set
`require driver.region == order.region`	Hard filter
`minimize currentLoad`	Lower is better
`maximize rating`	Higher is better
`explain`	Produce explanation

9.4 Preferences

A preference is softer than a requirement.

```
prefer driver.vehicle == "electric" weight 0.2
```

If the preference is not met, the candidate may still win if it is otherwise strong.

9.5 Phased problems

Large problems can be divided into phases.

```
problem ResolveIncident(incident: Incident) -> IncidentResolution:
    phase classify:
        severity = decide ClassifySeverity(incident)
        verify severity != none

    phase assignTeam depends on classify:
        choose team from Team where active == true
        require team.skills contains incident.category
        minimize team.currentLoad weight 0.5
        maximize team.successRate weight 0.5

    phase proposeAction depends on assignTeam:
        choose action from PlaybookAction where category == incident.category
        prefer action.previouslySuccessful weight 0.4

    verify:
        require assignTeam.result != none
        require proposeAction.result != none

    explain
```

Try it. Write a problem that selects the best warehouse for an order. Require enough inventory. Minimize distance. Prefer warehouses with low workload.

10 Decisions: Returning a Clear Answer

10.1 What is a decision?

A **decision** evaluates evidence and returns a typed result.

```
decision ApproveOrder(order: Order) -> Bool:
    evaluate order.total
    evaluate order.customer.riskScore

    require order.total > 0

    return order.customer.riskScore < 0.7

    explain
```

10.2 Problem versus decision

Construct	Best for
`problem`	Choosing the best candidate from many options
`decision`	Returning a clear result from evidence

10.3 Decision result structures

```
structure ApprovalResult:
    approved: Bool
    reason: Text
    riskScore: Decimal

decision ApproveVendor(vendor: Vendor) -> ApprovalResult:
    evaluate vendor.riskScore
    evaluate vendor.documentsComplete

    if vendor.riskScore > 0.8:
        return ApprovalResult:
            approved = false
            reason = "Risk too high"
            riskScore = vendor.riskScore

    return ApprovalResult:
        approved = true
        reason = "Approved"
        riskScore = vendor.riskScore
```

Try it. Write a decision that approves a refund when the order was delivered late and the customer account is active. Return a structure with `approved` and `reason`.

11 Plans, Steps, and Human Work

11.1 Why plans exist

A decision answers a question. A plan coordinates work.

Story. A vendor wants to join the company platform. The system can check documents, score risk, request human approval, notify finance, and activate the vendor. Some steps are automatic. One step requires a human reviewer. If a later step fails, earlier changes may need compensation.

11.2 Steps

A **step** is a reusable typed unit of work.

```
step VerifyInventory(order: Order) -> InventoryCheck:
    require order.product.inventory >= order.quantity

    return InventoryCheck:
        available = true
        remaining = order.product.inventory - order.quantity
```

11.3 Plans

```
plan ApproveOrder(order: Order) -> Solution<OrderApproval>:
    step inventory = VerifyInventory(order)
    step risk = ScoreRisk(order)

    step approve depends on inventory, risk:
        require inventory.result.available
        require risk.result.score < 0.7

        return OrderApproval:
            approved = true

    verify:
        require approve.completed

    trace full
```

11.4 Human steps

Some work must be done by a person.

```
human step ReviewVendor(vendor: Vendor) -> ReviewResult:
    assign to role "ComplianceReviewer"
    due in 2.days
    priority high
    needs vendor.securityDocuments
    returns ReviewResult
```

A human step can block a plan until the task is completed, approved, or rejected.

11.5 Partial solving

Partial solving means: do everything possible with the current information, then report what is missing.

```
result = partial solve ApproveVendor(vendor) trace full

if result.status == "partial":
    show result.missing
```

11.6 Compensation

Compensation describes how to undo or balance a successful step if a later step fails.

```
step ReserveInventory(order: Order) -> Reservation:
    transaction:
        require order.product.inventory >= order.quantity
        order.product.inventory -= order.quantity
        save order.product

    compensate:
        transaction:
            order.product.inventory += order.quantity
            save order.product
```

11.7 End-to-end plan story

Business step	Solve mapping
Check whether vendor uploaded documents	`step VerifyDocuments`
Score vendor risk	`step ScoreVendorRisk`
Ask compliance reviewer	`human step ReviewVendor`
Activate vendor if approved	`step ActivateVendor`
Undo activation if later failure occurs	`compensate`
Explain the whole process	`trace full`

Try it. Design a plan for onboarding a new employee. Include one automatic step, one human step, one verification, and one compensation block.

Part IV

Real-Time Runtime

12 Events: Handling the Outside World

12.1 What is an event?

An **event** is something that happened outside or at the boundary of the system.

Examples:

- Order received
- Payment captured
- Shipment delayed
- Sensor reading arrived
- Customer submitted document

12.2 Declaring an event

```
event OrderReceived:
    id: Text
    customerId: Text
    productId: Text
    quantity: Int
    receivedAt: Instant
```

12.3 Handling an event

```
on OrderReceived:
    transaction:
        create Order:
            externalId = event.id
            customerId = event.customerId
            productId = event.productId
            quantity = event.quantity
            status = "pending"
```

12.4 Why events are durable

If a system crashes after receiving an order but before processing it, the order should not disappear. In Solve service mode, events are recorded, dispatched, retried, and traced.

12.5 Algorithm-to-code mapping

Algorithm	Solve mapping
Define incoming message shape	`event OrderReceived`
Receive and store event	`solve event publish` or connector input
Validate fields	event schema
Process safely	`on OrderReceived` handler
Change data atomically	`transaction`
Explain processing	trace entry
Retry failures	supervised runtime

Try it. Create an event `PaymentFailed` and an `on PaymentFailed` handler that marks an order as `payment_failed` and creates a support ticket.

13 The Supervised Runtime

13.1 What `solve serve` does

The command `solve serve` runs the long-lived runtime. It supervises workers that process durable work.

```
solve serve --profile production
```

In production, the runtime coordinates:

- event workers
- reaction workers
- schedule workers
- plan workers
- connector retry workers
- human-task resume checks
- trace retention and verification

13.2 Why supervision matters

Story. Imagine a warehouse receives 5,000 order events. During processing, the machine restarts. A fragile script might lose work or repeat the same operation twice. A supervised runtime should recover persisted work, continue safely, avoid duplicate processing, and record what happened.

13.3 Runtime behavior

The supervised runtime provides:

- bounded queues
- backpressure
- retries with limits
- dead-letter queues
- idempotency checks
- graceful shutdown
- startup recovery
- health and readiness states
- metrics

- trace entries

13.4 Useful commands

```
solve serve --profile production
solve health
solve runtime status
solve runtime workers
solve event list
solve schedule list
solve plan list
solve trace search
```

Try it. Explain what should happen if a server crashes after an event is stored but before its handler finishes. Then write the event and handler for a simple `OrderReceived` flow.

14 Schedules: Time-Based Work

14.1 What is a schedule?

A **schedule** runs work based on time.

```
schedule CheckDelayedOrders every 15.minutes:
    partial solve ResolveDelayedOrders() trace summary
```

14.2 Common schedule uses

- Check overdue tasks every hour
- Recalculate risk every night
- Escalate incidents after 15 minutes
- Retry pending connector calls
- Generate daily operations report

14.3 SLA monitor example

```
schedule CheckSLA every 5.minutes:
    overdue = Ticket where status != "closed" and dueAt < Clock.now()

    for ticket in overdue:
        create Escalation:
            ticket = ticket
            reason = "SLA breached"
            priority = "high"
```

14.4 Operational behavior

Schedules in service mode should be durable. If the runtime is down during a due time, missed-run recovery can catch up according to configuration.

Try it. Create a schedule that checks unpaid invoices every day and creates a reminder task for invoices older than 7 days.

15 Connectors and Secrets

15.1 The connector idea

Solve should be able to connect to other systems without hardcoding thousands of product-specific integrations into the language.

A connector is configured, permissioned, limited, traced, and redacted.

15.2 Calling a connector

```
result = Connector.mail.sendEmail:
    to = customer.email
    subject = "Order delayed"
    body = "Hello \(customer.name), your order is delayed."
```

15.3 Connector configuration concept

A connector definition describes the system, operations, timeouts, limits, authentication, and response types. It can be loaded from configuration such as `Connectors/*.json`.

```
structure SendEmailRequest:
    to: Text
    subject: Text
    body: Text

structure SendEmailResponse:
    id: Text
    status: Text
```

15.4 Secrets

Secrets are values such as tokens and passwords. They should never appear in logs, traces, errors, or exports.

```
secret MAIL_API_TOKEN
```

15.5 Testing connectors

Use mocks to test connector-dependent code without sending real messages.

```
test "sends delay message":
    mock Connector.mail.sendEmail returns SendEmailResponse:
        id = "msg-001"
        status = "sent"

    result = solve NotifyDelayedCustomer(order)

    expect result.status == "complete"
```

Try it. Design a connector operation named `readCustomer`. Define request and response structures. Then write a test that mocks the operation.

16 Reasoning Traces and Observability

16.1 What is a trace?

A **reasoning trace** is a structured explanation of what the runtime did and why.

It can include:

- inputs
- entity reads and writes
- rules checked
- policies checked
- events processed
- connector calls
- plan steps
- human tasks
- candidate scores
- missing information
- verification results
- final result

16.2 Requesting a trace

```
result = partial solve ApproveVendor(vendor) trace full
show result.trace
```

16.3 Trace commands

```
solve trace search --category event.processing
solve trace export --format jsonl
solve trace replay <trace-id>
solve trace compare <trace-a> <trace-b>
solve trace verify
```

16.4 Why traces matter

Traces answer questions that ordinary logs often cannot answer clearly:

- Why was this driver selected?

- Which rule blocked approval?
- What document is missing?
- Which connector failed?
- What happened before a compensation step ran?
- Did a secret leak into output?

Try it. For a refund approval decision, list five facts that should appear in the reasoning trace.

Part V

Language Features for Real Systems

17 Units and Conversions

17.1 Why units matter

Unit mistakes cause real problems: cases versus pallets, kilograms versus pounds, hours versus minutes, rupees versus dollars, tickets versus labor hours.

Solve lets you define custom units and conversions.

17.2 Declaring units

```
unit Case:
    symbol "case"
    dimension Count

unit Pallet:
    symbol "plt"
    dimension Count
```

17.3 Simple conversion

```
conversion Case to Pallet:
    require value >= 0
    return value / 48
```

17.4 Contextual conversion

Some conversions depend on context.

```
conversion Case to Pallet for Product:
    require product.casesPerPallet > 0
    return value / product.casesPerPallet
```

17.5 Using units

```
cases = 240.case
pallets = cases as Pallet
```

Try it. Define units `Ticket` and `LaborHour`. Create a conversion or rate that represents 12 tickets per labor hour.

18 Text and Strings

18.1 Why text support matters

Real operational systems contain names, addresses, emails, incident notes, vendor documents, comments, CSV fields, API payloads, and explanations.

18.2 Common operations

```
name = "  Asha Rao  "
clean = name.trim().titleCase()
slug = clean.slug()
upper = clean.uppercase()
message = "Hello \(customer.name), your order \(order.id) is ready."
```

18.3 Redaction

```
emailHidden = Text.redact(customer.email, pattern: Pattern.email)
```

18.4 Escaping

```
csvSafe = Text.escapeCSV(input)
htmlSafe = Text.escapeHTML(input)
```

18.5 Parsing

```
amount = Text.parseDecimal(input)
date = Text.parseDate(input, format: "yyyy-MM-dd")
```

Try it. Write code that takes a customer's full name, trims spaces, converts it to title case, and creates a slug.

19 Packages and Modules

19.1 Why packages exist

Packages let teams share reusable code: math functions, industry rules, connector definitions, structures, policies, and tests.

19.2 Module declaration

```
module RedRuff.MathX

public function triple(value: Int) -> Int:
    return value * 3
```

19.3 Importing

```
import RedRuff.MathX

value = MathX.triple(10)
```

19.4 Package patterns

Good packages include:

- clear module names
- public API only where needed
- tests
- examples
- package capability notes
- versioned behavior

Try it. Create a package named `Company.Risk`. Add a public function `riskBand(score: Int) -> Text`.

20 Testing

20.1 Basic tests

```
test "triple multiplies by three":
    expect MathX.triple(4) == 12
```

20.2 Testing entities and problems

```
test "assigns available driver":
    driver = create Driver:
        name = "Asha"
        status = "available"
        region = "south"
        rating = 4.9

    order = create Order:
        region = "south"
        weight = 20

    result = solve AssignDriver(order)

    expect result.value.name == "Asha"
```

20.3 Testing partial solving

```
test "vendor waits for missing document":
    vendor = create Vendor:
        name = "ACME"
        documentsComplete = false

    result = partial solve ApproveVendor(vendor)

    expect result.status == "partial"
    expect result.missing.count > 0
```

20.4 Testing with mocks

```
test "connector is mocked":
    mock Connector.mail.sendEmail returns SendEmailResponse:
        id = "test-message"
        status = "sent"

    result = solve SendWelcomeEmail(customer)

    expect result.status == "complete"
```

Try it. Write a test for a rule that prevents negative inventory.

Part VI

Building Real-World Solutions

21 Build 1: Inventory Replenishment

21.1 The story

A warehouse sells products every day. Some products run out quickly. If the team waits too long, customers cannot buy. If the team orders too much, money is stuck in inventory. The system should watch stock, decide when to reorder, and create a plan.

21.2 Algorithm

1. Model products and inventory.
2. Preserve truth: inventory cannot be negative.
3. Detect low stock.
4. Choose supplier or warehouse action.
5. Create a replenishment plan.
6. Trace why the plan was created.

21.3 Coding map

Algorithm step	Solve feature
Model product and stock	`entity Product`, `entity Inventory`
Prevent invalid values	`rule`
Watch low stock	`when Inventory.onHand becomes ...`
Select action	`problem ReplenishInventory`
Coordinate work	`plan ReplenishmentPlan`
Explain	`trace full`

21.4 Code

```
entity Product:
    sku: Text
    name: Text
    casesPerPallet: Int = 48

entity Inventory:
    product: Product
    onHand: Int
    reserved: Int = 0
    reorderPoint: Int
    targetLevel: Int

rule InventoryNeverNegative:
    always Inventory.onHand >= 0
```

```
when Inventory.onHand becomes 0:
    partial solve ReplenishInventory(Inventory) trace full

problem ReplenishInventory(inv: Inventory) -> ReplenishmentAction:
    choose action from ReplenishmentAction where action.product == inv.product

    require inv.onHand <= inv.reorderPoint
    require action.quantity > 0

    minimize action.cost weight 0.5
    minimize action.leadTimeDays weight 0.3
    maximize action.supplierReliability weight 0.2

    explain
```

21.5 Exercise

Add a unit conversion from `Case` to `Pallet`. Then update the replenishment action to show both cases and pallets.

22 Build 2: Vendor Onboarding

22.1 The story

A new vendor wants to sell through the company. The business needs documents, risk checks, compliance approval, and final activation. Some steps are automatic. Some require humans. Some information may be missing.

22.2 Algorithm

1. Capture vendor details.
2. Verify required documents.
3. Score risk.
4. If risk is high, request human review.
5. Activate only after verification.
6. If activation fails, compensate earlier changes.
7. Keep a trace.

22.3 Code

```
entity Vendor:
    name: Text
    category: Text
    riskScore: Decimal
    documentsComplete: Bool = false
    status: Text = "draft"

structure VendorApproval:
    approved: Bool
    reason: Text

step VerifyVendorDocuments(vendor: Vendor) -> VerificationResult:
    needs vendor.documentsComplete when vendor.documentsComplete == false

    return VerificationResult:
        passed = vendor.documentsComplete
        reason = "Documents verified"

human step ComplianceReview(vendor: Vendor) -> ReviewResult:
    assign to role "ComplianceReviewer"
    due in 2.days
    priority high
    needs vendor.documentsComplete
    returns ReviewResult

plan OnboardVendor(vendor: Vendor) -> Solution<VendorApproval>:
    step docs = VerifyVendorDocuments(vendor)
    step review = ComplianceReview(vendor) depends on docs
```

```
    step activate depends on review:
        require review.result.approved

        transaction:
            vendor.status = "active"
            save vendor

        return VendorApproval:
            approved = true
            reason = "Vendor activated"

        compensate:
            transaction:
                vendor.status = "draft"
                save vendor

    verify:
        require vendor.status == "active"

    trace full
```

22.4 What the developer learns

This example combines documents, missing information, human work, durable plan state, compensation, and traceability.

Try it. Extend the plan with a finance review step for vendors whose expected monthly volume is above 1,000,000.

23 Build 3: Incident Response

23.1 The story

A production incident happens. The system needs to classify severity, assign the right team, choose a playbook action, and escalate if the SLA is at risk.

23.2 Algorithm

1. Model incidents, teams, and playbook actions.
2. Classify severity.
3. Select an active team with the required skill.
4. Prefer less loaded teams with high success rate.
5. Pick an action.
6. Schedule SLA checks.
7. Trace the reasoning.

23.3 Code

```
entity Incident:
    title: Text
    category: Text
    severity: Text = "unknown"
    status: Text = "open"
    openedAt: Instant
    dueAt: Instant

entity Team:
    name: Text
    active: Bool = true
    currentLoad: Int
    successRate: Decimal
    skills: Text[]

problem ResolveIncident(incident: Incident) -> IncidentResolution:
    phase classify:
        severity = decide ClassifySeverity(incident)
        verify severity != none

    phase assignTeam depends on classify:
        choose team from Team where active == true
        require team.skills contains incident.category
        minimize team.currentLoad weight 0.5
        maximize team.successRate weight 0.5

    phase proposeAction depends on assignTeam:
        choose action from PlaybookAction where category == incident.category
        prefer action.previouslySuccessful weight 0.4
```

```
    verify:
        require assignTeam.result != none
        require proposeAction.result != none

    explain

schedule CheckIncidentSLA every 5.minutes:
    overdue = Incident where status != "closed" and dueAt < Clock.now()

    for incident in overdue:
        create Escalation:
            incident = incident
            reason = "SLA at risk"
            priority = "critical"
```

Try it. Add a human step for manager approval when severity is `"critical"`.

24 Build 4: Customer Support Escalation

24.1 The story

A support ticket starts simple. Then the customer replies three times, the order is delayed, and the ticket becomes urgent. The system should escalate, notify a person, and explain why.

24.2 Algorithm

1. Model tickets and messages.
2. React when message count increases.
3. Decide priority.
4. Escalate if priority is high.
5. Notify through connector.
6. Trace the outcome.

24.3 Code

```
entity Ticket:
    customerEmail: Text
    status: Text = "open"
    priority: Text = "normal"
    messageCount: Int = 0
    orderDelayed: Bool = false

when Ticket.messageCount changes:
    result = decide ClassifyTicket(Ticket)

    transaction:
        Ticket.priority = result.priority
        save Ticket

    if result.priority == "high":
        Connector.mail.sendEmail:
            to = Ticket.customerEmail
            subject = "Your case has been escalated"
            body = "We are prioritizing your ticket."

decision ClassifyTicket(ticket: Ticket) -> TicketPriority:
    evaluate ticket.messageCount
    evaluate ticket.orderDelayed

    if ticket.orderDelayed and ticket.messageCount >= 3:
        return TicketPriority:
            priority = "high"
            reason = "Delayed order with repeated customer replies"
```

```
    return TicketPriority:
        priority = "normal"
        reason = "No escalation needed"
```

Try it. Add a schedule that checks tickets older than 24 hours and escalates them if still open.

Part VII

Operations and Production Use

25 SolveDB Operations

25.1 What SolveDB is for

SolveDB stores application entities and runtime state used by rules, reactions, events, plans, traces, and schedules.

25.2 Common commands

```
solve db verify
solve db backup --out Backups/today
solve db restore --from Backups/today
solve db recover --to-commit 184
solve db compact
solve db migrate --preview
solve db migrate
solve db rollback
```

25.3 Operational patterns

Situation	Use
Before migration	`solve db backup` and `solve db migrate --preview`
Database health check	`solve db verify`
Need to undo a safe migration	`solve db rollback`
Crash left incomplete WAL tail	`solve db recover`
Storage has grown after many changes	`solve db compact`
Need historical recovery point	point-in-time recovery command

25.4 Rules of thumb

- Always preview migrations before applying them.
- Verify backups, not just create them.
- Keep recovery procedures tested, not just documented.
- Use pagination or cursors for large reads.
- Treat operational commands as part of your release process.

Try it. Write a deployment checklist that includes database backup, migration preview, migration apply, verification, and rollback plan.

26 Performance, Scalability, and Concurrency

26.1 What performance means in Solve

Performance is not only raw speed. For real-time systems, performance also means predictable latency, bounded memory, safe concurrency, crash recovery, and no unbounded queues.

26.2 Validated release-gate highlights

Benchmark	Result
Parser throughput	about 53.6M chars/sec
Easy syntax lowering	about 5.46M/sec
SolveDB writes	about 226k/sec
SolveDB reads	about 2.78M/sec
One-million-record load writes	about 270k/sec
One-million-record paginated reads	about 3.46M/sec
One-million-load peak memory	about 3.08 GB under 4 GB threshold
Startup p95	about 20.5 ms
Studio API p95	about 25.4 ms

26.3 Designing scalable Solve programs

- Keep candidate sets bounded before solving problems.
- Use indexed fields for common filters.
- Prefer pagination for large reads.
- Avoid connector calls inside tight loops unless rate-limited.
- Use schedules for periodic checks instead of constant polling.
- Use trace summary mode for high-volume routine paths and full trace for critical paths.
- Use idempotency keys for external effects.

26.4 Concurrency patterns

Solve service mode uses supervised workers, bounded queues, and durable state. Your source should still be written carefully:

- Put multi-record changes inside transactions.
- Use requirements to prevent invalid state.
- Make external effects idempotent.

- Keep compensation steps safe to retry.
- Do not assume a reaction or connector call runs exactly once.

Try it. A reaction sends an external email. Explain why idempotency matters. Then design an idempotency key for the email.

27 Errors, Retries, and Dead-Letter Queues

27.1 Kinds of failure

Real-time systems fail in many ways:

- invalid input event
- rule violation
- connector timeout
- schedule body failure
- reaction error
- human rejection
- compensation failure
- database recovery situation

27.2 Retry and DLQ pattern

A durable worker should retry temporary failures and move poison messages to a dead-letter queue after bounded attempts.

```
reaction NotifyDelayedCustomer
when Order.status becomes "delayed":
    Connector.mail.sendEmail:
        to = Order.customer.email
        subject = "Order delayed"
        body = "We are working on it."
```

If the connector fails temporarily, the worker can retry. If it fails repeatedly, the work becomes inspectable in the DLQ instead of disappearing.

27.3 Production debugging workflow

1. Inspect the failed event, reaction, schedule, connector, or plan step.
2. Read the trace.
3. Check redacted error details.
4. Decide whether to replay, retry, compensate, or mark as handled.
5. Add a test for the failure if it represents a domain case.

Try it. Describe how you would handle a connector timeout that happened while sending a customer notification.

28 Deployment and Release Checklist

28.1 Development cycle

```
solve init MyProject
solve check
solve test
solve run
solve serve --profile development
```

28.2 Production checklist

1. Run tests.
2. Run package verification.
3. Preview migrations.
4. Backup and verify database.
5. Apply migrations.
6. Run database verification.
7. Start `solve serve` with production profile.
8. Check health and readiness.
9. Verify event, reaction, schedule, connector, and trace metrics.

28.3 Release honesty

A production release should clearly state what was validated and what was not. For example, a Windows x86-64 release should not claim cross-platform validation unless those artifacts were actually built and tested.

Try it. Write a release note template that lists platform, build command, tests, benchmark summary, known limitations, and verification hash.

Part VIII

Reference and Practice

29 Developer Cheat Sheet

Modeling

```
entity Product:
    name: Text
    inventory: Int

structure Result:
    ok: Bool
    reason: Text
```

Create

```
product = create Product:
    name = "Keyboard"
    inventory = 12
```

Query

```
items = Product where inventory < 10
   sort by inventory limit 50
```

Transaction

```
transaction:
    require product.inventory > 0
    product.inventory -= 1
    save product
```

Rule

```
rule NonNegative:
    always Product.inventory >= 0
```

Reaction

```
when Order.status becomes "paid":
    solve FulfillOrder(Order)
```

Problem

```
problem SelectDriver(order: Order) ->
    Driver:
    choose driver from Driver where
   status == "available"
    require driver.region == order.
   region
    minimize driver.currentLoad
   weight 0.6
    maximize driver.rating weight 0.4
    explain
```

Plan

```
plan ApproveOrder(order: Order) ->
   Solution<OrderApproval>:
    step risk = ScoreRisk(order)
    step approve depends on risk:
        require risk.result.score <
   0.7
        return OrderApproval:
            approved = true
    trace full
```

Event

```
event OrderReceived:
    id: Text
    quantity: Int

on OrderReceived:
    create Order:
        externalId = event.id
        quantity = event.quantity
```

Schedule

```
schedule CheckSLA every 5.minutes:
    partial solve
   ResolveOverdueTickets() trace
   summary
```

Connector

```
Connector.mail.sendEmail:
    to = customer.email
    subject = "Hello"
    body = "Welcome"
```

Test

```
test "works":
    expect 1 + 1 == 2
```

30 Capstone Exercises

30.1 Capstone 1: Library borrowing system

Build a small system that manages books and borrowers.

Requirements:

- Model `Book`, `Borrower`, and `Loan`.
- Prevent negative copies.
- React when a loan becomes overdue.
- Decide whether a borrower can borrow another book.
- Add a schedule to check overdue loans daily.
- Add tests.

30.2 Capstone 2: Delivery dispatch

Build a dispatch system.

Requirements:

- Model orders, drivers, and regions.
- Select the best driver with a problem.
- React to incoming order events.
- Send a connector notification.
- Trace the decision.
- Add a failure test for no available driver.

30.3 Capstone 3: Vendor onboarding

Build a plan with human review.

Requirements:

- Model vendors and documents.
- Use partial solving for missing documents.
- Include a human compliance step.
- Add compensation for activation.
- Add trace export.
- Add tests for approved, rejected, and partial cases.

30.4 Capstone 4: SLA operations monitor

Build a scheduled incident monitor.

Requirements:

- Model tickets and escalations.
- Use a schedule to scan due items.
- Use a decision to classify priority.
- Use a connector to notify the responsible team.
- Use DLQ handling for notification failure.
- Add a trace search example.

31 Glossary

Term	Meaning
Entity	Persistent business object stored by the application
Structure	Typed value used for results, payloads, and temporary data
Rule	Condition that must always remain true
Requirement	Condition that must be true for a specific operation
Reaction	Domain behavior triggered by state change
Event	External or boundary input processed durably
Problem	Candidate selection with requirements, goals, preferences, and explanation
Decision	Typed answer based on evaluated evidence
Plan	Durable strategy made of steps and dependencies
Step	Reusable typed unit of work
Human step	Step that requires a person to act
Partial solve	Solve as far as possible with current information
Trace	Structured explanation and provenance of execution
Connector	Configured connection to an external system
Secret	Protected value such as token or password
Schedule	Time-driven work declaration
Compensation	Safe reversal or balancing action after a later failure
DLQ	Dead-letter queue for work that failed too many times
PITR	Point-in-time recovery for database state

Final Note

Solve is designed to make serious real-time software easier to build and easier to understand. Its power does not come from clever syntax alone. It comes from one consistent model:

> **State is explicit. Truth is protected. Change is handled. Choices are explained. Work is durable.**

Start small. Model one entity. Add one rule. Add one reaction. Then add a problem, a plan, a trace, and a connector when the domain asks for them.

That is how Solve stays simple at the beginning and powerful in production.

Index