

1. Introduction to Avaloq

Purpose of this student book is to provide understanding of Avaloq softwares and how they are customized according to the clients, for learning we have different different ways like we have

- Presentation slides
- Student Books
- Documents (docs.avalog.com)

Avaloq Certified Customization Professional (ACCP)

Avaloq Banking Suit (ABS) is a fully integrated banking solution. This ABS has

- Front - Middle - Back office functionality.

2. Functional Overview

A core banking system needs to support all aspects of business like

- to store all data safely (financial & client data)
- To keep detailed records

The system needs to communicate with various systems inside and outside of the bank.

★ It must follow all rules and regulations.

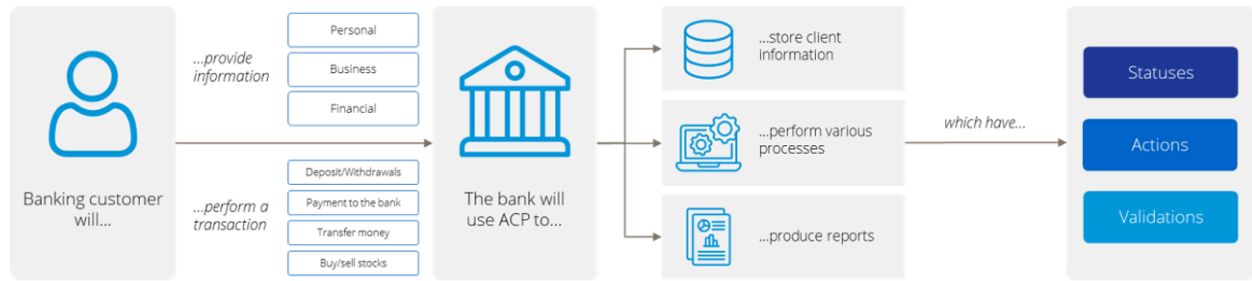
2.1 ABS

The ABS is complete suit of products offered by Avaloq for banking and wealth management. It has set of standards, localized preconfigured processes and a range of mission-critical third-party applications and adapters Including the Avaloq Front Platform and web and mobile banking.

2.2 ACP (Avaloq Core Platform)

The bank's job is to handle whatever services customers need—like managing accounts, processing transactions, and keeping their information safe. When a customer gives their details or makes a request, the bank records that data and carries out the transaction. All this information gets stored in the system's database.

Every service follows a specific process depending on how the bank operates. On top of that, the system (ACP) can also generate reports whenever needed, based on what the user asks for and the filters they apply.



In this diagram :

- Customer will provide information (- Personal - Business - Financial)
- Customer can perform transactions like : - Deposit / Withdrawal - Payment to bank - Transfer Money - Buy / sell stocks.
- After this bank will use ACP to - store client data - perform various processes - produce reports.
- Which will have statuses , Actions and Validations.

Smart Client

- This is the main user interface of ACP model.
- This is for all types of user and employees of a bank.

Front Office : - The front office is the part of a business that directly talks to customers and clients. These are the people who handle communication, build relationships, and stay in touch with clients.

Usually, the front office includes people like sales staff and finance team members. Their main job is to deal with customers and bring money into the company.

★ So basically, the front office is the “customer-facing team” that helps the company earn most of its income.

Middle Office : - The middle office mainly helps and supports the front office.

While the front office deals with customers and makes deals, the middle office makes sure everything is done correctly behind the scenes.

Their work includes:

- Checking and managing risk (making sure the company doesn’t take bad or risky deals)
- Calculating profit and loss
- Making sure all transactions are properly recorded and processed

So basically, if the front office “makes the deals,” the middle office “checks, controls, and records” those deals to make sure everything is safe and correct.

Back Office : The back office is the part of the bank that handles all the behind-the-scenes work. These employees don't talk to customers and they don't directly bring in money.

Instead, they take care of important support tasks like:

- Managing paperwork and records
- Supporting daily business operations
- Making sure the bank follows laws and rules (compliance)

Even though they don't earn money directly, the back office is very important because without it, the bank wouldn't run properly or legally.

2.3 Avaloq Front Platform

The AFP caters to both the needs of front office employee as well as client.

Front Workplace (FWP) & Web banking provides these services making use of web based technology and design.

FWP

The FWP is a web-based interface used by front office employees, to be able to manage clients via an interface which provides functions relevant to their role and is optimized for front related tasks.

FWP aims to make a client-facing employee's life easier.

Web and mobile banking

Web and mobile banking are self-service interfaces whose users are clients of a bank. Through this, a client can execute transactions on their own or view their portfolio through the online portals of the bank.

★ The functionality provided by web and mobile banking is much more limited than those provided by the SmartClient or the FWP,

since they are only catering for clients of the bank (external user).

Summary

Bank requires a comprehensive IT solution because it performs many different different functions.

The **front office** of a bank deals directly with customers.

the **middle office** provides support for this

the **back office** provides underpinning functions such as security, without which the bank couldn't function.

The **ABS** provides a solution which can be used by front, middle and back-office bank employees as well as for self-service banking by clients.

Graphical User Interface

Smart client is the Avaloq's GUI

Smartclient Interface

- SmartClient has “desks”, in which they can organize their client relations, manage security accesses, create reports, or monitor the processes that are running.
- SmartClient GUI is divided into different areas:

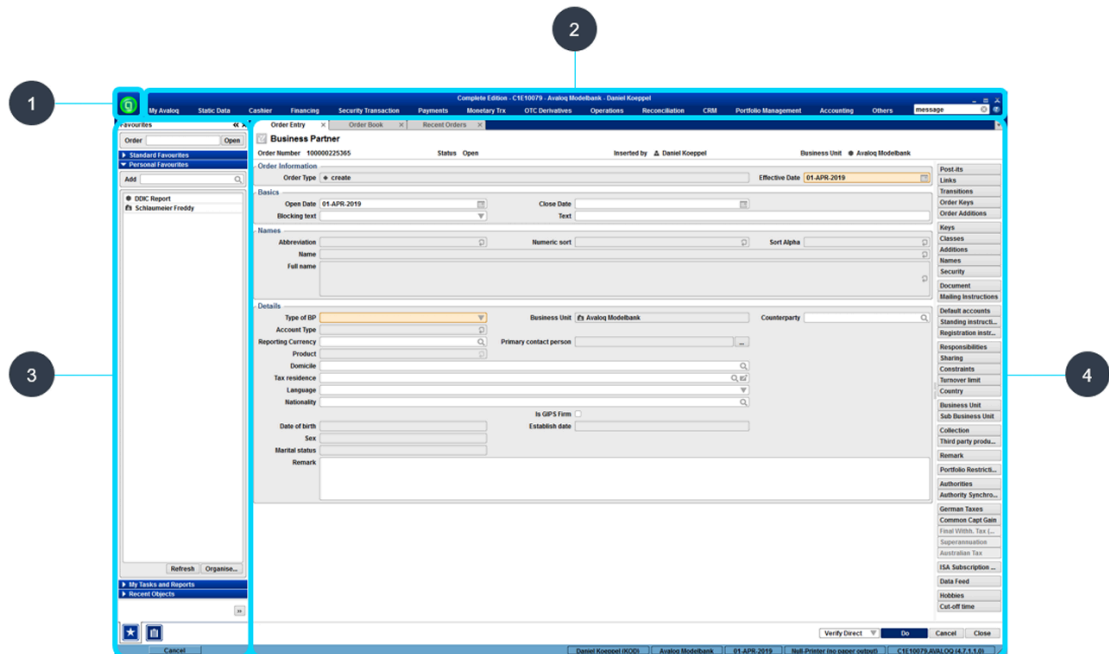
Menus

Sidebar

a search bar for lookups

a forms space

a favourites area for personalized quick access.



- **Avaloq Button** : The Avaloq button provides access to the Avaloq menu (click the button to open the menu).
- **Ribbon** : The upper section of the main window is called the ribbon.

- **Sidebar** : The sidebar gives you an efficient, personalized access to overviews (such as reports or orderbooks) and to workflow actions, via context actions.
- **Work area** : Forms will be open in this work area.

Order forms allow you to perform most of the activities necessary in the Avaloq Banking Suite, such as:

- Create and modify objects
- Search for and view object details
- Perform transactions
- View reports and orderbooks

On the right-hand side, there are subwindows that you can open by clicking a button.






At the top of the screen, next to the order form, you'll see different tabs like the order book and recent orders. You can keep one order form open and also open multiple reports at the same time.

At the bottom of the SmartClient, there's a status bar that shows details like the date, the database you're connected to, and the default printer.

BU (Business Unit)

This represents an instance of bank or banks different different branches.

Form Fields

Field icons	Description
	The choice field opens a drop-down list of predefined values. These are also called drop-down list fields , since they are based on a concept called code tables which will be discussed later in the course.
	The lookup field is used to search for objects. When the icon is clicked, it will open a Lookup form.
	The date field allows you to select date values from a calendar or input certain date expressions.
 	The auto-manual switch fields are those whose values are derived or automatically filled when certain conditions are met. The field can be toggled from automatic calculation to manual edit mode. <ul style="list-style-type: none"> ▪ The former is the calculated mode, the field appears grey, and the value is automatically calculated ▪ The latter is manual mode where a user can instead manually edit the field.

Wild Card Searches :

- %
- *
- ?

Date Filed and its Expressions

In date fields, you can also input certain date expressions. Below are some examples:

Expression	Description
today	current date (date from the bank's point of view)
sysdate	system date (date from the database point of view)
+n	This is the current date plus <i>n</i> number of calendar days. E.g. +7 will produce the date value of current date plus seven calendar days.
+n v	This is the current date plus <i>n</i> number of bank days. E.g. +7v will produce the date value of current date plus seven bank days.
+n w +n m +n q +n s +n y	Adds <i>n</i> number of a time period: <ul style="list-style-type: none"> ▪ w (for week) ▪ m (for month) ▪ q (for quarter) ▪ s (for semester) ▪ y (for year)
sow som soq sos soy	start of current time interval: <ul style="list-style-type: none"> ▪ w (start of current week) ▪ m (start of current month) ▪ q (start of current quarter) ▪ s (start of current semester) ▪ y (start of current year)
eotw eotm eotq	End of the current time interval: <ul style="list-style-type: none"> ▪ w (end of the current week) ▪ m (end of the current month) ▪ q (end of current quarter) <p>Take note that there are no equivalents for the semester and year.</p>
mindate maxdate	The minimum and maximum date

DESKS

TASK AND REPORTING DESK

This is helpful for creating task template, setting up parameter values for tasks and can be used to search and execute tasks.

This can be accessed from the Avaloq GUI (Smart Client)

- **Report search** – used for search criteria like specific keywords or task type.
- **Search result** – displays the list of task templates returned by the search.
- **Subscribed** – lists the task templates that a user has subscribed to.
- **All** – lists all the task templates available
- **Task template** – for setting parameters before running a task.

- **Button area** – used for executing or modifying a task template.

Central Service Desk

It's an administration toolbox that provides the functionality to manage and control the Avaloq core. It helps user to do technical resources management such as output management, monitoring and logging and background processing.

4 Object model and orders

An object is like a thing, in banking there are many things and those can be different different too.

For ex : - Bank client

- Address
- contact

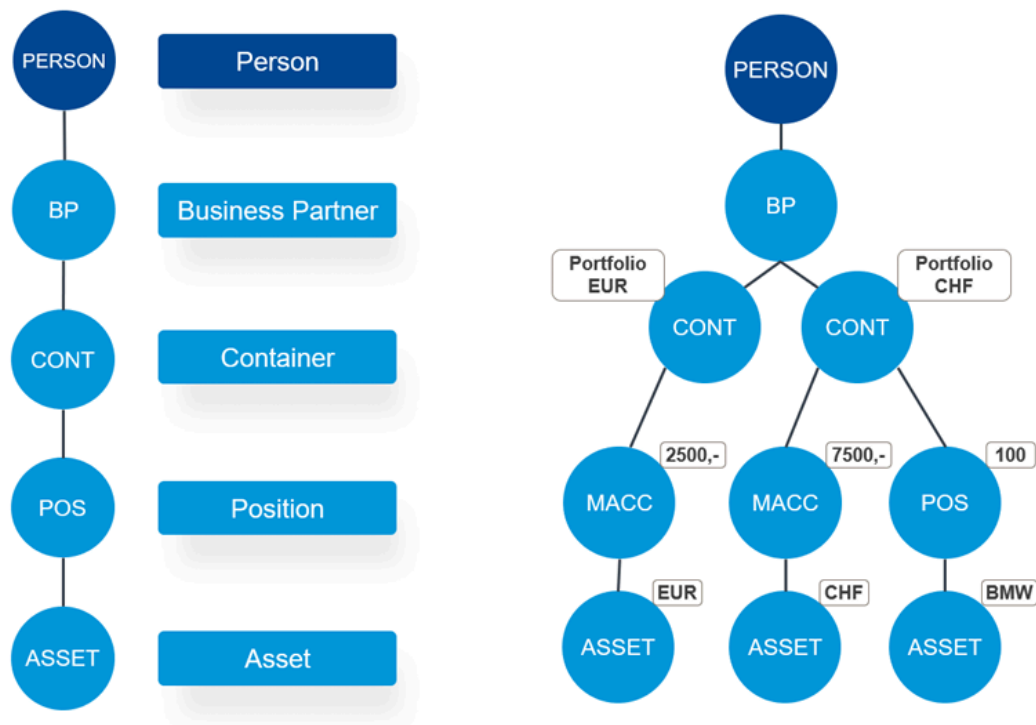
So basically, Avaloq breaks real-life banking info into small “things” (objects) to manage them easily.

You don't need to worry about how data is stored in the database.

You just work with simple “objects” instead of complex data structures.

It makes banking operations easier to understand and use .

4.1 Object model



This diagram shows how a bank organizes all information and accounts for a single client in a structured way.

At the top level is the **person (client)**, which includes personal and financial details collected when they apply to open an account. This information forms the basic client profile.

Next comes the **business partner layer**, which represents the formal relationship between the bank and the client. It includes internal details that the bank maintains, such as client IDs and “for bank use only” data.

Below that is the **container (portfolio)**. A portfolio acts as a holder or organizer for all the client’s financial products. A client can have one or more portfolios, often separated by currency or purpose (for example, EUR or CHF portfolios).

Inside each portfolio are **positions or accounts**. These represent specific financial products, such as savings accounts, each associated with a balance or value.

At the lowest level are the **assets**, which are the actual holdings—such as money, investments, or other valuables—linked to those accounts.

Overall, the model shows a top-down structure:

Client → Bank relationship → Portfolio → Accounts/positions → Assets, illustrating how a bank manages and organizes a client’s financial data and products.

Person (PERSON)

The **Person** is basically a way to represent any real-world entity that the bank deals with. It’s not just limited to an individual human—it can cover different types of entities.

There are three main types:

- **Natural person:** This means an actual human being, like a regular customer.
- **Legal person:** This refers to entities that have legal standing, like companies, corporations, trusts, or partnerships. These can own accounts and enter into agreements just like a person.
- **Person association:** This is used for groups or entities that don’t have full legal status on their own, such as informal partnerships or small business setups.

Business Partner

A Business Partner represents the relationship between the bank and an entity (like a person, company, or group). While the Person object is about who or what the entity is in real life, the BP is about how that entity is connected to the bank.

For example, a person object might represent an individual named Rahul, but the BP object represents Rahul as a customer of the bank. So, the BP only exists when there is an actual relationship with the bank.

BP can also represent internal relationships within the bank itself, such as different branches, business units, or counterparties. In such cases, a BP might exist without being directly tied to a person.

An important idea here is that Person and BP are kept separate:

- Person = real-world identity
- BP = business relationship with the bank

The connection between them defines different roles or ownership types:

- **Account Owner (AO):** The person who legally owns the account or relationship.
- **Registered Owner (RO):** The main official owner recorded in the system. Each BP must have exactly one RO.
- **Beneficial Owner (BO):** Someone who benefits from the account, even if they are not the legal owner.
- **Authorities:** These are people who don't own the account but are allowed to perform actions like transactions on behalf of the owner.

Container and Position

It is basically a place where all the financial holdings of a business partner are stored and organized. This can include things like money, investments (securities), gold, or any other valuable assets.

Everything inside a container is called a **position**. So, a position is just an individual item or holding within that portfolio.

In simple terms, the container is like a folder, and the positions are the items inside that folder.

Assets

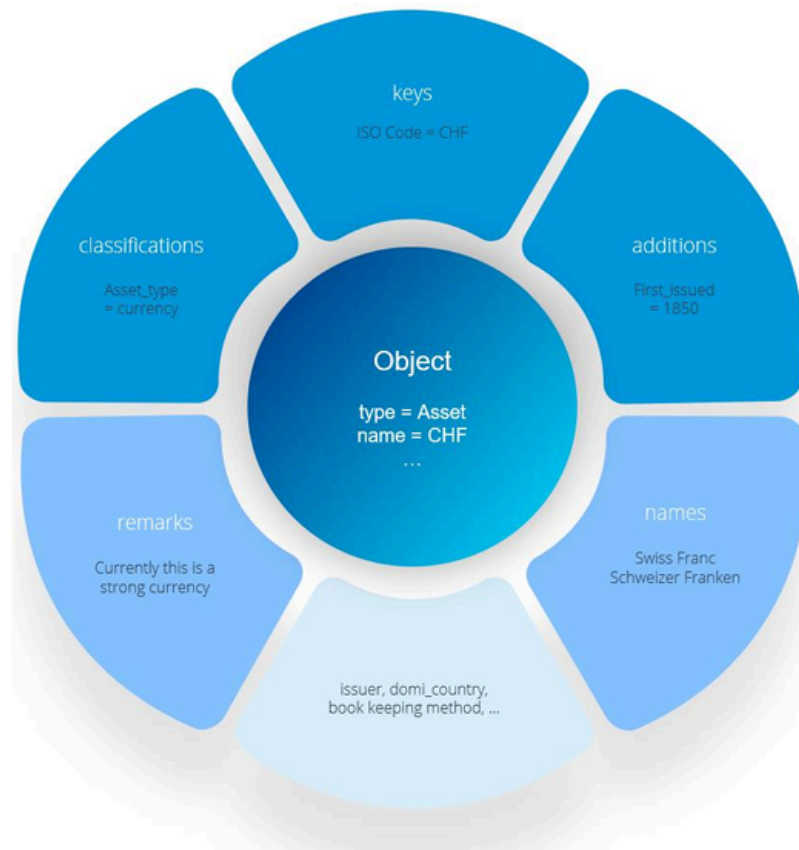
Asset (ASSET) represents the actual financial product itself. This could be things like currencies (money), stocks, bonds, or commodities like gold. It defines *what the item is*, along with its general properties, such as its value or other banking-related details.

An important distinction here is:

- The **Asset** is the *type of thing* (for example, USD, a specific stock, or gold).
- The **Position (or account)** is *how much of that asset is held* in a specific portfolio (container).

So in simple terms, the asset is the item itself, while the position tells you the quantity of that item a client owns in their portfolio.

Object attributes and extensions



Keys

Keys are basically identifiers used to quickly find and access objects in the system.

They act like shortcuts or labels. Instead of searching through everything, you just enter a key and the system directly locates the exact object.

A key can be:

- Something simple and easy to remember (like a customer's initials or a short name)
- Or something unique and official (like a customer ID or social security number from another system)

Keys can be:

- Manually created by users
- Or automatically generated by the system

In short, keys are just quick reference IDs that help in fast searching and retrieval of data.

Additions

Additions are basically extra/custom information that you can attach to any object.

They are flexible fields where you can store things like text, dates, numbers, or even choose from predefined options. This allows users to capture any additional details that aren't part of the standard data.

The main purpose of additions is to help users with decision-making by giving them more context or notes. They are not mainly used by the system for processing or calculations.

In simple terms, additions are like custom notes or extra fields added to an object for better understanding, not for core system operations.

There are two types of additions :

- Single-value addition : Which relates to holding single value as addition like BP's religion.
- Multi-value addition : Which relates to holding multiple values as addition like BP's hobbies.

Classifications

Classifications are like categories that the system uses to group objects.

The main difference from additions is that classifications are actually used by the system for processing and calculations. This means the system behaves differently depending on the classification.

For example, if a Business Partner (BP) is classified as:

- Employee
- Corporate customer
- Private customer

Then the system might:

- Charge different fees
- Apply different rules
- Process things differently

So in simple terms, classifications help the system decide what to do.

Classes

Classes are the possible values inside a classification.

Think of it like this:

- Classification is the category name
- Classes are the options inside it

Example:

- Classification: Customer Type
- Classes:
 - Retail customer
 - Employee
 - Private customer

Keys: used to find something quickly

Additions: extra information or notes

Classifications: categories that affect system behavior

Classes: the options inside those categories

Orders (Simple Explanation)

Orders are basically how actions happen in the system.

Whenever you want to:

- Create something
- Change something
- Or do a financial transaction

You use an **order**.

So in simple terms:

Orders = actions or operations performed in the system

Types of Orders

There are two main types:

1. Static Data Orders

These are used when you are working with **data only**, not money.

You use them to:

- Create objects (like a new customer)
- Update or modify existing data

Example:

- Creating a new Business Partner
- Changing a customer's address

So:

Static data orders = create or update information

2. Booking Orders

These are related to **financial transactions**.

Whenever money is involved, booking orders are used.

Example:

- Payment transactions
- Charges or fees

These orders also **trigger bookings**, which means:

- The system records the financial entry
- Accounts get updated

So:

Booking orders = money-related actions + system entries

What Both Types Have in Common

Both static data orders and booking orders include:

- **Workflows**
Steps or processes the system follows (like approvals or sequence of actions)
- **Extensions**
Extra custom features or fields added based on business needs

- **Validation**
Checks to make sure the data is correct
Example: making sure required fields are filled
- **Security**
Controls who is allowed to perform the order

Bookings (Simple Explanation)

Bookings are basically the **record of financial transactions** in the system.

Whenever money or assets move, the system creates a **booking** to keep track of it.

So in simple terms:

Bookings = recording of money movement

What is a Ledger?

A **ledger** is like a **book where all transactions are recorded**.

Earlier:

- Traders used to write transactions in a physical book

Now:

- Everything is stored digitally in the system

So:

Ledger = place where all bookings are stored

Why Bookings Are Important

Banks need to track:

- Money coming in
- Money going out
- Assets they own

This helps them:

- Know their profit
- Manage risks
- Prepare financial reports

How Bookings Happen in the System

When a financial transaction happens:

- A **booking order** is created
 - It is processed by something called the **Book Engine**
 - Then the system records the transaction as a **booking**
-

What Do Bookings Affect?

Bookings change **positions**.

A position is basically:

- How much money or asset you have

Example:

- If you receive money → your position increases
- If you pay money → your position decreases

Debit and Credit (Very Important Concept)

Every booking follows a rule called **double-entry bookkeeping**.

There are always two sides:

Debit (–)

- Source of the value
- Means money is coming out

Credit (+)

- Destination of the value
- Means money is going in

Important Rule

Debit must always equal Credit

This means:

- Every transaction has two entries
- Nothing is created or lost magically
- Everything stays balanced

Simple Example

If you pay 100:

- Your account → Debit (-100)
- Other account → Credit (+100)

Both sides match, so the system stays balanced.

Business Types

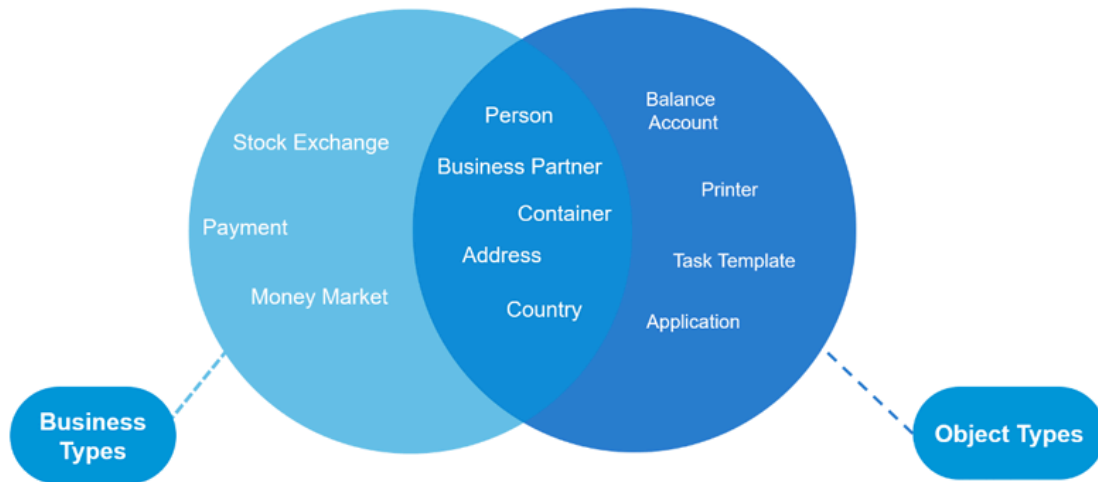
Business types are basically labels that are attached to every order to describe what that order is actually doing. They help the system understand the purpose of the order, whether it is related to a financial transaction or to creating or modifying data.

For booking orders, the business type tells you what kind of financial transaction is happening. For example, if the business type is "Payment," it means the order is used to transfer money into or out of an account. If the business type is "STEX," it means the order is related to a stock exchange transaction. So in simple terms, for financial actions, the business type explains what kind of transaction is being performed.

For static data orders, the business type tells you what kind of object is being created or updated. For example, if the business type is "BP," the order is used to create or modify a Business Partner. If it is "Address," then the order is working with address data. So here, the business type is basically telling you what kind of data you are dealing with.

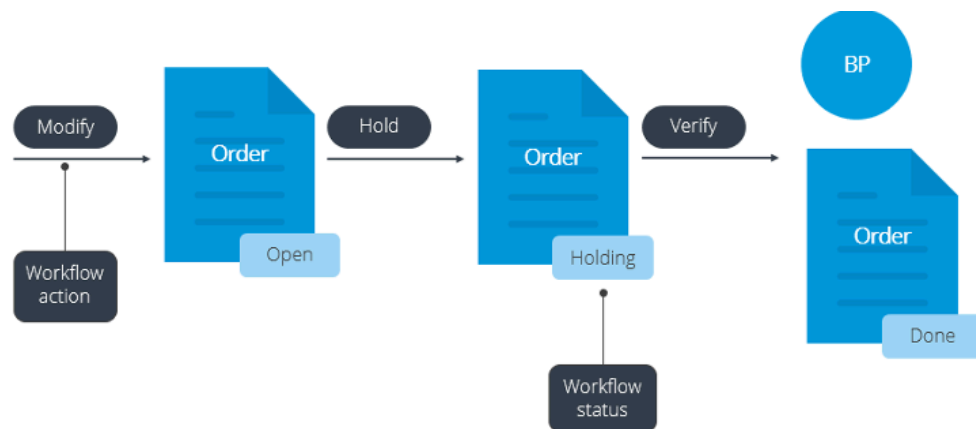
Every object in the system has something called an object type, which defines what kind of object it is, like a person or a business partner. In many cases, there is a matching business type used to create or update that object. For example, a person object is created using a person business type, and a BP object is created using a BP business type.

However, not all objects work this way. Some objects exist only for technical or internal system purposes, and they cannot be created or modified using orders. These are called technical objects, like task templates or printers. They are part of the system setup but not directly handled through business-type orders.



Workflow

Workflow is basically the journey an order goes through in the system. Every order is always in some status, which just means its current stage at that moment. From there, different actions can happen that move the order to the next stage. So you can think of it like this: status tells you where the order is right now, and actions are what push it forward. Step by step, the order keeps moving through different statuses until it reaches the final stage. In short, workflow is just the path an order follows from start to finish.



Order attributes and extensions

Orders are described using attributes, which are basically all the details that define an order. Some of these attributes are fixed by the system, while a small part called **extensions (EXTN)** can be customized.

These extensions mainly include **keys and additions**.

Additions are just for extra information or notes—they don't really affect how the system works, they're just there to store useful details. Keys are used to store reference data, especially when connecting with third-party systems or for data extraction.

Apart from extensions, there are some common attributes like **post-its**, which let users add notes to an order,

and **links**, which help connect and group related orders together.



Customization In Avaloq

Customization in Avaloq is basically about adjusting the standard system to fit how a specific bank works. Even though most banks offer similar services, their internal processes differ, so the Avaloq Banking System (ABS) needs to be tailored for each client. This tailoring is called customization, and Avaloq provides multiple ways to do it.

The system is built using two parts: kernel code and custom code.

Kernel code is the standard, out-of-the-box code provided by Avaloq and is common for all banks,

while **custom code** is where bank-specific changes and requirements are implemented.

This separation helps banks modify what they need without affecting the core system.

Customization can involve choosing which features of the system to use, extending existing functionality, or even building new features.

To make things easier, Avaloq provides something called a model database, which includes both the standard kernel and some example custom code. Instead of starting from scratch, banks can simply modify this existing setup based on their needs. During implementation, this model database is customized so the system behaves exactly as required by the bank. The people who handle all these changes are called customization specialists.

Sources

A source is basically a place where all customization code is written and stored. You can think of it like a document or file that contains logic (mostly in PL/SQL) used to modify how the system behaves. These sources are not static—they can be edited, versioned (so changes are tracked), and transported between systems. In simple terms, a source is just a container of customization code.

You can view and manage sources in two main ways. One is through the Task & Reporting Desk, where you can search using “Source list: .DEFAULT” (Task ID: 909). The other is through Avaloq ICE, which is the main development environment where customization specialists actually write and manage code. So, one is more like a system view, and the other is a developer workspace.

Source Owner and Namespace

Every source has an owner. It can either belong to Avaloq or to the client. Kernel sources are owned by Avaloq and are part of the standard system, which means you cannot modify them directly. On the other hand, client (custom) sources are created by the bank to implement their specific requirements.

To avoid confusion between different sources, Avaloq uses something called a namespace. A namespace is just a unique prefix (usually 2–4 letters) added to the source name, like `MDB$SOURCE_NAME`. This helps differentiate client code from standard Avaloq code and avoids naming conflicts. You can think of it as a name tag for your code, making sure everything stays organized and unique.

Override and Replacement (Core Concept)

Since kernel sources cannot be edited directly, customization is done by creating separate client sources. There are two main ways to do this: override and replacement.

Override (Partial Change)

Override is used when you only need to change some parts of the existing logic. In this case, you create a custom source that modifies specific fields or behavior while keeping the rest of the kernel logic unchanged. During compilation, the system merges the kernel source with your override source.

- Only specific parts are changed
- Remaining logic stays the same
- Final result = merged version of both sources

Change only what you need

Replacement

Replacement is used when the system does not allow partial changes, usually because the logic is too complex. In this case, you must create a custom source that completely replaces the kernel source. Even if you only want to change one line, you still need to rewrite the entire source.

- Entire kernel logic is replaced
- Kernel source is ignored during compilation
- Custom source must include everything

Rewrite everything from scratch

Customization Techniques

The Avaloq Banking System is built on top of an Oracle database, where all the data is stored in many connected tables. Any feature in the system must be written in a way that Oracle understands, like PL/SQL or data manipulation statements that can insert, update, or delete data.

However, directly accessing database tables from custom code is not a good idea. The reason is simple: if something changes in the database structure, like a table name or column name, then all the custom code using it can break. To avoid this risk, Avaloq provides different customization techniques that sit on top of the database and hide these details. This way, even if the database changes internally, the custom code keeps working.

These techniques include Avaloq Script packages, DSL sources, APIs, and rule loaders.

Avaloq Script packages

Avaloq Script packages are very similar to PL/SQL packages. They allow you to define variables, functions, and procedures. The syntax also looks very close to PL/SQL, and in the end, it gets converted into PL/SQL when compiled.

The main advantage of using Avaloq Script instead of plain PL/SQL is that it supports something called data dictionary expressions. These make it much easier to access data without worrying about which exact database table stores it. So, instead of writing complex SQL queries, you use a simpler and more abstract way to read or write data.

This abstraction ensures that even if the underlying database structure changes, the code still works without modification.

Using DSL sources

A DSL source contains code written in a special Avaloq-specific language designed for a particular part of the system. Since core banking systems are complex, there are many DSL types, each focusing on a specific domain.

Each DSL has its own syntax and its own compiler. The compiler checks if the code is correct, converts it into something Oracle understands like PL/SQL, and manages dependencies between sources.

These DSL sources define how different parts of the system behave. Even though they have different syntaxes, they are often connected and depend on each other. For example, order workflows and order validations are defined using different DSLs but still work together.

Using Avaloq APIs

APIs in Avaloq are basically PL/SQL functions or procedures that are used to perform operations in a controlled and secure way. Instead of directly writing SQL statements to change data, you use these APIs.

The main purpose of APIs is to provide a safe way to modify data, allow complex logic when Avaloq-specific languages are not enough, and enable external tools to interact with the system.

One big advantage is that you do not need to understand the full internal code of an API. You only need to know its name and required inputs to use it correctly.

There are three types of APIs. Secured APIs are provided by Avaloq or partners and are usually hidden. Non-secured APIs are generated from Avaloq Script and remain readable. Client-written APIs are created by the bank for specific needs.

Although APIs may be slightly slower than direct SQL because they include extra checks, they are much safer and more reliable.

Using rule loaders

Many system behaviors in Avaloq are controlled by rules. These rules follow simple if-then logic, like applying charges based on customer type. Different rule engines handle different areas, such as cost calculation or booking logic.

Rules are stored in special tables called rule loaders. These tables are not directly used during execution. Instead, their data is loaded into internal rule tables using APIs, which then control the system behavior at runtime.

This creates another level of abstraction. You only need to understand how to define rules in the rule loader, not how the internal system processes them. It also ensures that changes made during design do not affect the system until the rules are properly loaded.

Using native Oracle code

It is also possible to customize the system using direct PL/SQL or SQL statements. However, this approach should only be used as a last option.

Direct database access is risky because it depends on the internal structure of the database, which can change. That is why Avaloq recommends using higher-level customization techniques whenever possible, as they are safer and more stable.

Oracle SQL Developer

Oracle SQL Developer is a tool that helps you work with an Oracle database in an easy and user-friendly way. Instead of writing everything manually, it gives you a visual interface where you can see and manage database objects and also run code like SQL and PL/SQL.

With this tool, you can perform many common database tasks. You can browse existing tables to see what data is stored, create new tables, modify them, or delete them if they are no longer needed. It also allows you to run SQL queries and scripts to fetch or update data.

Another useful feature is working with PL/SQL. You can write, edit, and even debug your PL/SQL code directly in the tool, which makes development much easier.

Apart from that, it also lets you export data from the database and create reports for analysis.

Avaloq ICE

Avaloq ICE (Integrated Customization Environment) is the main development tool used for customizing the Avaloq system. It acts as a central workspace where developers can create, organize, store, and maintain code such as Avaloq Script and DSL sources. It also works as a code repository, keeping all changes managed in one place. Additionally, Avaloq ICE compiles

the written code into a format that Oracle can understand (like PL/SQL) and integrates with other Avaloq tools to handle all coding-related tasks efficiently from a single environment.

Avaloq ICE is a graphical user interface based on Eclipse and serves as the main development environment for Avaloq customization. It functions as an IDE where developers can create, compile, and maintain customization code, while also acting as the standard source code management and revision control system for the Avaloq Banking System. Unlike external tools such as GitHub or SVN, it includes its own central repository where all source code is stored and organized. As part of the Avaloq Tools Suite, it also supports tasks like integration testing (to check whether changes impact existing code), creating installation packages (so changes can be applied easily), and managing ACP/AFP databases.

Avaloq ICE is made up of several components.

The ice Workbench is the actual IDE installed locally,

while the ice workspace stores user-specific settings, indexes, and temporary files.

The ice server (repository) is the central database that stores all source code for the Avaloq community, mainly for storage purposes.

The ice client represents a client-specific area (such as for a bank) within the repository, containing its customized code.

Finally, the instance is the actual database where the system runs, holding business data and compiled (deployed) sources.

An important point is that the repository and instance store code differently. When code is saved in ICE, it is stored in the repository, but it only appears in the instance after compilation. Similarly, if something is deleted or reset in the instance (like compiled code), it does not affect the repository. This separation ensures that source code remains safe even if the runtime system is reset.

Editor – Area in which source contents are modified. Depending on the type of source being edited, the appropriate editor is displayed in the editor area. Main types: textual, graphical, form based.

Views – These are subwindows which present information other than editing, for example navigation views or instance information. A view might appear by itself or be stacked with other views in a tabbed notebook.

Perspectives – A defined set of views and editors appropriate to different customization areas. There are several predefined perspectives. These can be modified and saved to use again.

Orderbook view – A view that allows organization, management, and query of different entities in Avaloq ice. An entity is part of source management to properly store and organize codes in

the repository. Orderbook is an expression (query) to easily access the entity in the repository. For example, you can use the USEROPEN(ME) expression to an entity owned by you in order to access this in the orderbook view, including any sublevels if there are any.

Remember: this entity exists in the repository, but you won't see it unless you use a search expression which targets it.

Change

A Change in Avaloq ICE is the smallest unit used to install code. You cannot install single sources—only a change that contains one or more sources. However, you can compile individual sources for testing. Each change is linked to exactly one task.

Source

In Avaloq ICE contains the actual code. It is identified by a name and type, and it is versioned. Sources are created or checked out within a change, and each source belongs to a specific component.

Component

Is a logical group used to organize related sources, for example by function, team, or organization. Each component has an owner who is responsible for reviewing the sources created within it.

Release

in Avaloq ICE is a package that groups multiple changes and source versions together for deployment. It creates an installable software package that is usually deployed once to the production instance.

Example: If developers create 20 changes for a banking feature update, all those changes are collected into one release package and then deployed to the live banking system together.

Avaloq ICE Stream

An ICE Stream is a logical branch that organizes source versions in order during development. It represents a separate development path and contains at least one release.

Avaloq Releases and Fix Packs

- **Release** = major software package with new features or changes.
- **Fix Pack** = smaller update containing bug fixes or minor improvements.
These are accessed using the Delivery Tool.

Source Deployment

Source deployment means converting Avaloq sources into executable program code (mostly Oracle PL/SQL).

There are **2 deployment methods**:

Remote Compilation

- Quickly deploys a single source to the preferred instance set in ICE Workbench.
- Mainly used during development/testing.
- Works only for source types with their own compiler.
- Compiles only the current source.

Installer / Installation Package

- Deploys all sources inside a change together.
- Compiles sources in the correct installation sequence.
- Also recompiles dependent sources.
- Used for full and proper deployments.
- Slower than remote compilation.

Note :Some source types like **BIN** and **DOKU** are not deployed because they are only used for setup or documentation and contain no executable behavior.

Source Integration

Before a source becomes part of the Avaloq repository, it must be properly tested, checked-in, and integrated. The developer first tests the source locally by deploying it through remote compilation or an installer. After successful testing, the source is checked-in to the repository, where it becomes available for review by a code auditor, although some reviews may be automated. Once the audit is approved, the source goes through integration testing. If integration succeeds, the changes become available in the repository for other developers to check out. Finally, the changes enter the release cycle and are deployed to the bank's production instance.

Deployed Sources

The Deployed Sources view in Avaloq ICE shows all sources currently deployed and active in an instance. These sources may come from remote compilation, installation packages, or standard Avaloq database sources. You can only check out a source if it also exists in the ICE

repository, because the instance stores executable code while the repository stores the actual source code and versions.

This will also show you the status of the sources which can have one of the following statuses: •

Valid – Source has been correctly compiled and can be used in the system

• Invalid – When a source is compiled, dependent sources may be tagged as invalid. Or when a new source cannot be compiled due to some dependencies, it is also tagged with this status. Any functionality tied to an invalidated source will not work properly.

• Dropped – Sources marked for deletion. Dropped sources will be removed by a special task in the end of day process if there are no references to them. • All other statuses “Ready for...” and “To be installed” appears if a source needs a re compilation because of a change (typically via remote-compilation) of related sources.

- In the Instance Information view, you have access to the Avaloq Invalids and Avaloq Invalids Errors tabs. This shows all sources that have been invalidated or impacted as a result of any of your deployments to your instance.

Source Code Repository

The Source Code Repository view shows all sources that are stored in the ICE repository. These include sources that have been integrated, checked-in, or created by other users who have access to the same repository. It helps developers view and manage shared source code available for development and version control.

Avaloq Solution Model Development (ASMD) and Context

The Avaloq Solution Model Development (ASMD) helps developers create better customization code more efficiently. It provides useful features such as code validation, syntax checking, formatting, code suggestions, and reference resolution. These features help detect errors early during development.

Many ASMD features work only inside an active Context. A Context is a container that connects development work to a specific ABS instance (database environment). It contains tasks, changes, and sources, and validates the source code against that instance.

When working with a Context, the code is checked and validated locally while typing, which saves time and reduces installation errors. It also provides content assist, which gives coding suggestions during development.

Without a Context, source code must be compiled and validated directly on a remote ABS instance. In this case, some errors may not be detected early, and installing unchecked code can sometimes cause issues that may even require re-cloning the database.

Model Cache in ASMD

When a Context is active and a developer performs an action that needs source information, ASMD uses the Model Cache. The Model Cache stores precomputed metadata about sources from an Avaloq Core instance.

ASMD first checks the local Model Cache to quickly find the required information. If the data is not available locally, it requests the information from the remote Model Cache. The result is then stored locally so it can be reused faster in future queries.

Cloning and Restore Points

Cloning and restore points are used to manage the database and recover from major issues.

Cloning a new image completely resets the database to its original state, as if no changes were ever made. All created objects, orders, and deployed sources are permanently removed. Because it deletes all testing work and takes a long time, it should only be used as a last resort.

A **restore point** creates a snapshot of the database at a specific time. It saves all objects, orders, and deployed sources existing before that timestamp. The restore point can be used for 24 hours and allows developers to quickly roll back changes if errors occur. It is faster and safer than re-cloning the database, especially during heavy development and compilation work.

Data Dictionaries (DDICs)

Data Dictionaries (DDICs) are an intermediate layer between Avaloq Script and database tables. Instead of directly accessing database tables or data, developers use DDICs to retrieve the required information. DDICs work like a directory that collects and organizes data from different tables, making data access simpler, safer, and easier to manage.

Data Dictionaries (DDICs) Concepts

Data Dictionaries (DDICs) allow Avaloq Script to access different types of data, such as objects and orders, whether stored in the database or loaded in memory. They act like a map to database data, giving read and sometimes write access without using SQL statements.

DDICs hide the complexity of database tables, so developers only need to know DDIC syntax instead of the underlying table structure. This abstraction makes data access simpler and more consistent.

Because of this design, DDIC-based data access remains stable even if the underlying database tables change.

Object	OBJ_%	This layer stores permanent object data in the database. The data is updated only after the related order is fully processed and verified. For example, if a person's marital status is changed from "single" to "married" and sent for approval, the object still shows "single" until the order is verified. After verification, the new value is saved in the object layer.
	OBJ_BP	

Data Access Layer	Examples	Simple Explanation
Order Memory	MEM_DOC_% MEM_DOC_BP MEM_DOC_PAY	This layer stores order data temporarily in memory while a user is working on it. When a user opens an order form, the data is loaded from the database into memory. Any changes made are stored here until the user saves or cancels the order.
Order Persistence	DOC_% DOC_BP DOC_PAY	This layer stores orders permanently in the database after they are saved. It includes completed orders and also unfinished orders that are saved for later review or approval. It also stores details such as who created the order and the current workflow status.

Structure Of a DDIC

Fields

Fields store information about the DDIC target, such as an object or an order. The type of information depends on the DDIC. For example, a BP can contain fields like address, domicile, and language, while a payment can contain fields like amount, beneficiary, or ordering client. Fields can store different kinds of data such as text, numbers, dates, time values, booleans, IDs, object references, or lists of values.

Methods

Methods define actions that can be performed on the DDIC target. They describe what the DDIC can do. For example, a method in the `mem_doc` DDIC allows developers to add a post-it note to an order.

Constructors

Constructors are used to create new items such as orders, files, or messages in Avaloq Script. They are available only in the order memory layer DDICs.

Avaloq Script

Avaloq Script is the programming language used by customization specialists to customize Avaloq systems. It is similar to PL/SQL, but the main difference is how data is accessed. In PL/SQL, data is accessed using SQL queries, while in Avaloq Script, data is accessed through DDICs instead of direct SQL. In simple terms, Avaloq Script can be understood as PL/SQL without SQL.

Avaloq Script Basics – Structure

Avaloq Script can be used in many source types, but `SCRIPT PACKAGE` sources are mainly used to define procedures and functions. A procedure is used to create, update, or recalculate something, often by using methods and constructors. A function is used to return a value that can later be used in other places such as procedures, forms, or reports.

Naming Conventions

When creating script packages in Avaloq Script, developers use naming conventions to keep the code organized and easier to maintain. Avaloq Script is usually written for a specific business context such as BP, PAY, or STEX, so functions and procedures are grouped according to that business type.

Script packages are also organized based on DDIC layers. The `mem_doc` layer is used for temporary order data in memory, the `doc` layer is used for persisted order data, and the `obj` layer is used for object data.

The naming format is:

- `<namespace>$mem_doc_<business type>` for memory layer orders, for example `acd$mem_doc_bp`

- <namespace>\$doc_<business type> for persisted orders, for example acd\$doc_bp
- <namespace>\$obj_<object type> for objects, for example acd\$obj_bp

Using variables Note –

Names of variables and parameter in Avaloq Script follow a convention where:

“l_” stands for local variable (inside a procedure or function)

“b_” for body variable (inside the whole script package)

“c_” for constants (constant values like thresholds, dates, etc.)

“i_” for input parameter (variable to be filled while calling)

Code Tables

Code tables are database tables that store predefined or allowed values used in the Avaloq Banking System (ABS). Like normal tables, they contain rows and columns, and each column has a data type such as text, number, or date.

The main purpose of code tables is to control what values can be entered into certain fields. Instead of allowing free text, the system only accepts values stored in the code table. This ensures consistency and reduces errors.

For example, values like payment channels, notification types, object classifications, error messages, and order keys are stored in code tables.

In simple terms, code tables act like a centralized list of valid options used across the system.

Code Table Sources

In Avaloq, code tables are configured using two source types: **CODE TAB DEF** and **CODE TAB**.

CODE TAB DEF (Code Table Definition) is used to define the structure of the code table. It specifies things like:

- which columns the table has,
- whether columns are mandatory or optional,
- the data type of each column (text, number, date, etc.),
- and which parts can be customized later.

CODE TAB (Code Table Data) is used to define the actual data stored in the table. Each row represents one entry, and the values for each column are filled in here.

When these sources are compiled:

- the system creates the physical database table using the structure from the CODE TAB DEF source,
- fills the table with the rows defined in the CODE TAB source,
- and automatically generates a DDIC so developers can access the table data indirectly in Avaloq Script instead of using SQL directly.

In simple terms:

- **CODE TAB DEF = table structure**
- **CODE TAB = table data/entries**

Numeric and symbolic identifiers

Each row in a code table must have a unique identifier so the system can distinguish one row from another.

Earlier, Avaloq used **numeric identifiers**. Developers had to assign numbers manually, and certain number ranges were reserved for the Avaloq kernel while others were available for customer customizations. This required developers to carefully track which numbers were allowed.

To simplify this, Avaloq introduced **symbolic identifiers**, where rows are identified using meaningful text instead of numbers. For example, instead of using an ID like **1001**, a row can use a name like **purple** or **xyz\$custom_colour**.

Symbolic IDs have two main advantages:

- developers no longer need to manage number ranges,
- and the code becomes easier to read and understand because the identifier clearly describes the row's purpose.

Internally, Avaloq still automatically assigns a numeric ID in the database, but developers mainly work with the symbolic ID.

If a symbolic ID has no namespace or starts with **avq\$**, it usually belongs to the Avaloq kernel. For custom developments, best practice is to use your own namespace, such as **xyz\$custom_colour**.

Another best practice is to keep symbolic IDs short and meaningful so they remain readable in Avaloq Script.

To check whether a code table uses numeric or symbolic identifiers, developers look at the **CODE TAB DEF** source. Most modern kernel and custom code tables now use symbolic identifiers.

Standard code table parameters

In a code table, each row contains values for different columns. These values are defined using **parameters**. Each parameter represents the value of one specific column in that row.

The available parameters are defined in the **CODE TAB DEF** source, because it specifies the structure of the code table and its columns. Since different code tables can have different columns, the number and type of parameters can vary from one code table to another.

However, many code tables share some common standard parameters, such as:

- `user_id` → a user-friendly identifier,
- `descn` → description text,
- `intl_id` → internal identifier.

So, when viewing code tables, you will often see these standard parameters repeated across many different tables.

Parameter	Description
<div>activ</div> <div>inactive</div>	<p>The status of the code table entry. The default is activ.</p> <p>Example:</p> <pre>row xyz\$custom_colour inactive user_id "CUSTOM_COLOUR" descn "This is a custom colour"</pre>
intl_id	<p>The intl_id (internal identifier) is a key primarily used to identify the entry in the CODE TAB source. It has the following properties:</p> <ul style="list-style-type: none"> ▪ It is a text string ▪ Must be all lowercase. ▪ For code tables that use symbolic IDs, the intl_id value is implicitly the symbolic ID value. ▪ Kernel defined intl_id cannot be overridden
user_id	<p>The user_id is also a key used to identify the entry in the CODE TAB source. Unlike the intl_id, the user_id is visible in the user interface and is used so that a user (a bank employee) can quickly find the entry from the list of values. It has the following properties:</p> <ul style="list-style-type: none"> ▪ It is a text string ▪ Must be all uppercase ▪ Must not contain any namespaces as this is a customization-specific term. A bank employee will not know what “MDB\$” or “XYZ\$” will mean. ▪ Kernel defined user_id can be overridden
name	A text string that defines the name of the code table entry.
descn	A description of the code table entry.

Code table definition (CODE TAB DEF)

This defines structure of Code Table, what columns will a code table have

Key Column : A key column is a column used to uniquely identify each row in a table and retrieve data from it. It usually contains internal ID values such as intl1_id or intl2_id. The key column is mandatory and every value in it must be unique, meaning no two rows can have the same ID. Most tables use intl1_id as the key column, while intl2_id is mainly used in partitioned tables, which are tables divided into smaller sections for better management and performance.

Private and public range

This section in the CODE TAB DEF defines several things and has the following syntax:

- private range undefined rows
- public range undefined rows

Originally, Avaloq used numeric IDs to identify rows in the CODE TAB table. These IDs were divided into two ranges:

a private range and a public range.

The private range was reserved for the system's internal kernel use,

while the public range was available for customization specialists to create and manage their own custom entries.

With the introduction of symbolic IDs, we no longer need to keep track of these numeric ranges. There are, however, two attributes we need to take note of:

Attribute	Description
Range key type	<p>This defines if symbolic IDs can be used for the code tables or a combination of numeric and symbolic IDs. The possible values can be:</p> <ul style="list-style-type: none"> ▪ ident: rows will make use of symbolic IDs ▪ number_and_ident: rows can make use of symbolic IDs and numeric IDs for the private range ▪ number_or_ident: the same as above but for the public range ▪ numeric/number: in some legacy sources, this can be found followed by the range of numbers. Note that there are some sources where it is simply just the range of numbers without this declaration. <p>Example:</p> <pre>private range ident <action> undefined rows public range ident <action> undefined rows</pre>
Action	<p>Specifies what the compiler does to the database table when a row is removed from the CODE TAB source.</p> <p>It can have the following values:</p> <ul style="list-style-type: none"> • delete: deletes the row in the database table if it is removed in the source • deactivate: mark the row in the database table as inactive if it is removed from the source. • ignore: the row will be unchanged in the database table even if it is removed from the source. • enforce: (obsolete) throws an error. <p>Example:</p> <pre>private range ident deactivate undefined rows public range ident deactivate undefined rows</pre>

If a CODE TAB DEF source **has no public range** line, then customization specialists **cannot add** any rows to the code table.

```
code table code_pay_chan definition

  key_column intl_id

  private range ident    enforce undefined rows
  public  range ident    enforce undefined rows
  other   range          enforce undefined rows

  naming "PAY_CHAN"
  package def pay_chan
  override any_status any_row

  column intl_id      text 30      no_override
  column user_id      text 15      null  override
  column name         text 80      null  override
  column descn        text 400     null  override
  column meta_typ_list text 40      null  override

end code table
```

Override

This setting defines what changes can be made to kernel rows. It can include these options:

- **any_status** → Allows changing a row's status to **active** or **inactive**.
- **any_row** → Allows modifying any row. However, which columns can be changed depends on the column definitions.
- **any_column** → Allows changing values in **CODE TAB DEF** columns, except the **key_column**, which cannot be changed.

If no override option is given, the system automatically uses **override any_status** by default.

Column definition

The column definition has the following syntax:

`column <column name> <data type> <nullability> <override setting> <check>`

Column property	Description
Column name	The name of the column
Data type	<p>The following are the data types available:</p> <ul style="list-style-type: none">• number• boolean• date• text• reference (a reference to data in other code tables) <p>The data type text can be followed by the max character length.</p>
Nullability	<p>Defines if a column can be nullable or not.</p> <ul style="list-style-type: none">• If a column can be nullable, then the null keyword is used.• If the null keyword is not declared, the column is mandatory and cannot be nullable.
Override setting	<p>Defines if the column can be overridden or not. Available values are:</p> <ul style="list-style-type: none">• no_override: column cannot be overridden• override: column can be overridden. <p>If the override setting is not declared, then by default the column can be overridden.</p>
Check	<p>Specifies if there is a check being made on the column. The check keyword is used followed by a PL/SQL function or block.</p> <p>Example:</p> <pre>column intl_id text 10 check ":intl_id = lower(:intl_id)"</pre> <p>In the example above, the check enforces the intl_id column to have lowercase characters.</p>

The **Code Table Data Dictionary (DDIC)**

is updated whenever code table data sources are compiled. These DDICs are used in Avaloq Script to access code table values. The DDIC name is always the same as the code table name.

- The field names in the DDIC are the same as the **intl_ids** in the code table, such as:
 - `sic`
 - `eurosic`
 - `swift`
- You can access column values using this pattern:
`<code table>.<intl_id>.<column name>`

Examples

- To get the numeric ID of an entry:
 - `code_pay_chan.sic_id`
 - Similar SQL query:

```
select id from code_pay_chan where intl_id = 'sic'
```
- To get the name of an entry:
 - `code_pay_chan.sic.name`
 - Similar SQL query:

```
select name from code_pay_chan where intl_id = 'sic'
```

Extra Note

- For easier customization, the DDIC also stores numeric identifiers in fields ending with `_id`, such as:
 - `sic_id`
 - `swift_id`

Partitioned code table sources

Some code tables can contain thousands of entries, making the CODE TAB source very large and difficult to maintain. To solve this, the entries can be divided into smaller groups called **partitions**. These groups are usually based on:

- Object type
 - Business type
 - Component
-
- Partitioning helps organize large code tables into smaller and easier-to-manage files.
 - Whether a code table can be partitioned depends on the **CODE TAB DEF**.

- A code table supports partitioning only if the keyword **partition** is defined in the CODE TAB DEF, followed by the partition criteria.

Naming Convention

Each partition is stored in a separate CODE TAB source using this format:

`<code table name>.<partition criteria>`

This makes large code tables simpler to maintain and organize.

Contribution

are used to split large CODE TAB sources into smaller files for easier maintenance. Unlike partitions, they do not have any business meaning and are only for structural organization. The keyword **contributed by** defines how the table is split, and the specified column controls the split logic. In the example, the table is divided using `env_compo_id`, which can also be null, allowing a central CODE TAB source to exist. The source names must match the `intl_id` values from `CODE_ENV_COMP0`. Contributions can also be combined with partitions for further splitting. For contributed tables, the symbolic key must remain unique within a partition or within the whole table if no partitions exist.

Rule engines

are important customization areas in the ABS because many system behaviors are controlled by rules that are processed at run-time. Different rule engines handle different tasks. For example, the Cost Engine calculates costs for financial transactions, the Booking Engine decides how transactions should be booked, and the Balance Engine calculates the bank's balance sheets.

Rule basics

Fundamentally, a rule does the following:
IF {criteria} are met THEN {result} For example:

- If {payment currency = CHF} then {payment channel := SIC}
- If {order currency = EUR} then {dispo bank := Deutsche Bank}
- If {bank domicile country = Switzerland} and {asset = bank guarantee} then {value date := trade date plus 3}

- If {order type = internal payment} then {destination bank BP := model bank} You also need a rule which matches if none of the given conditions match. We call this a “catch all” – rule: • If {(no criteria)} then {payment channel := SWIFT}

Rule Customization

Each rule engine has its own **rule tables**, identified with the suffix _RULE (for example, PAY_CHAN_RULE). These are operational tables that are continuously used by rule engines at run-time, so they cannot be modified directly. To safely customize rules during design-time, Avaloq provides **rule loader tables**, identified with the suffix _RULE_LD (for example, PAY_CHAN_RULE_LD). These tables are updated using SQL statements or API procedures.

Advantages of Rule Loader Tables

- You do not need to know the internal structure of operational tables.
- The interface remains stable even if operational tables change between releases.
- Customizations can therefore be retained more easily after upgrades.

Different rule loader tables exist for different rule engine areas in the ABS

Rule Sets

are groups of rules inside a rule loader table. Each rule set contains multiple rules that define different outcomes based on conditions. For example, one rule set may check a value and decide different results depending on ranges (less than 10, between 10 and 20, greater than 20).

Each rule set has a **unique name** within a rule loader table. In Avaloq systems, rule sets can be organized in different ways depending on the table. In most cases, they are grouped by **business unit**, such as a rule set like DLFT_UK in BE_BOOK_RULE_LD for UK-specific rules. In some cases, they are grouped by **business area or type**, such as PAY in DOC_DFLT_FLD_VAL_RULE_LD, which contains rules for payment-related processes.

Rule Loading

Rule loading It is the rule set that you load from a rule loader table into a rule table, not the individual rules. When you load a rule set, you specify the name of the rule set and all the rules in that set are loaded in an atomic operation – either all the rules in the set are loaded or none of them are loaded. This means that if there is an error in one of the rules in the rule set, then no rules are loaded. Otherwise, all the rules in the rule set are loaded. If the rule set has been loaded before, all the rules in it will be deleted and completely replaced with the newly loaded rules from the rule set. If the rule set is loaded successfully, then it is considered activated. There are two ways to load the changes from the rule loader table to the operational rule table:

- Using the rule_Id# API. This is the method that a customization specialist must use.

- A rule loader task, which can be executed via the Task and Reporting Desk.

Rule Loader Table Structure

In Avaloq, rule loader tables are structured into five column groups called sections: Naming, Matching Criteria, Valuation Constraint, Results, and Miscellaneous/Admin. Each section serves a different purpose in defining how rules are identified, when they apply, what conditions they check, and what outcome they produce.

The exact columns in each section, and their meaning or syntax, depend on the specific rule loader table being used. This means every rule loader table can have a slightly different structure, even though the overall five-section concept remains the same.

Below is the explanation for the sections of a rule loader table and some examples of the columns in that section.

Section	Column	Description
Naming	RULE_SET	<p>The name of the rule set.</p> <p>The RULE_SET column is standard for all rule loader tables.</p>
Matching criteria	<p>Examples:</p> <p>BP_CLASS</p> <p>CURRY</p>	<p>These are the columns that define the matching criteria for the rule. Once the conditions defined in these columns are met, then a corresponding result is executed.</p> <p>The columns that can be used as a matching criteria depend on each rule loader table.</p> <ul style="list-style-type: none"> ▪ BP_CLASS: is a column in COST_RULE_LD where you can define a BP's class value as a matching criteria (e.g. if BP is classified as Employee in the Customer Type classification). ▪ CURRY: is a column in PAY_CHAN_RULE_LD where you can define a currency as a matching criteria (e.g. if the payment order has a currency of USD).
Valuation constraints	<p>Examples:</p> <p>PRIO</p> <p>VALID_FROM</p> <p>VALID_TO</p>	<p>These columns define the constraints for the rule, such as the priority of the rule or its validity period.</p> <p>The PRIO, VALID_FROM, and VALID_TO are standard for all rule loader tables.</p>
Results	<p>Examples:</p> <p>DEST</p> <p>PAY_CHAN</p>	<p>These are the columns that will define what the result would be when the defined matching criteria are met.</p> <p>The columns that define the results depend on each rule loader table.</p> <ul style="list-style-type: none"> ▪ DEST: is a column in BE_BOOK_RULE_LD which defines what the destination account would be for a matched booking rule. ▪ PAY_CHAN: is a column in PAY_CHAN_RULE_LD which defines what payment channel would be the result of a matched payment channel rule.

Matching criteria in Avaloq rule loader tables define when a rule should be triggered. A rule only executes its result if **all matching criteria are satisfied**, meaning the conditions are combined using **AND logic**. For example, if META_TYP = PAY and CURRY = CHF, then the

rule applies only when both conditions are true, and it may produce a result like PAY_CHAN = SIC.

If a matching criteria column is empty or undefined, it means any value is accepted for that condition, so it does not restrict the rule. A rule with no matching criteria at all is called a **catch-all rule**, which always executes and acts as a fallback when no other rules in the rule set match.

There are some matching criteria columns where the value that you can put in it are references to object keys or object classes.

Reference by	Syntax	Description
Object key	<key type>:<key value> Example: bp_sym:DIOBR	<p>If you want to set a specific object as a matching criteria, use the key type intl_id and the key value. For example:</p> <p>In this case, bp_sym is the intl_id of the key type "Symbol" and DIOBR is the key value for a BP object "Dio Brando".</p> <p>Details about the key type, including its intl_id, are stored in the CODE_OBJ_KEY table.</p> <p>Details about the key value of a specific object are stored in OBJ_REL_KEY.</p>
Object class	<classif>:<class> Example: bp_custr_type:custr	<p>If you want to set a class value as the matching criteria, use the intl_id of the classification and class. For example:</p> <p>In this case, bp_custr_type is the intl_id for the classification "Customer Type" and custr is the "Customer" class.</p> <p>Details about the classification and the class, including its intl_id, are stored in the CODE_OBJ_CLASSIF and CODE_OBJ_CLASS, tables respectively.</p>

In Avaloq rule loader tables, numeric IDs like #455676 or #205:8270 can be used to reference objects or classifications, but customization specialists are not allowed to use this format.

Some matching criteria columns also support multiple conditions in one field. This is done using separators:

- A comma (,) means AND
- A semicolon (;) means OR

For example, in COST_RULE_LD, the column BP_CLASS can be written as:

- `bp_custr_type:custr, bp_branch:zh` → means both conditions must be true (Customer AND Zurich branch)
- `bp_custr_type:custr; bp_custr_type:emp` → means either condition can be true (Customer OR Employee)

However, not all columns support multiple clauses, so you must always check the specific rule loader table documentation before using this feature.

Validation Constraint

In Avaloq rule loader tables, **valuation constraints** control which rule is chosen when multiple rules match. This is done using the **PRIO (priority)** column. The rule with the highest priority is executed first.

If more than one rule matches and they have the same priority, the system may randomly pick one, which can lead to unpredictable results. Because of this, priorities must be carefully designed.

A **catch-all rule** should always have the lowest priority so it only runs when no other rule matches. If it is given a higher priority, it may override other valid rules and prevent them from being executed.

Loading rule sets via an API

Recall that there are two ways to load rule sets to activate the rules, one is using an API and the other is using a task and executing it via Task and Reporting Desk.

Avaloq Model Bank (MDB)

in Avaloq is the standard base setup used for implementing new banking projects and for clients running the Avaloq Core Platform on an **on-premise** model, meaning each client customizes an existing bank setup rather than building from scratch.

The MDB is a long-established framework made of images, guidelines, processes, and documentation developed over many years. It serves as the baseline for most client installations (over 80%) and for official documentation. It is also a familiar environment for most developers and is continuously updated to lead new Avaloq software releases.

Avaloq Banking Reference/Starter Kit (ABR/S)

in Avaloq is the standard baseline for new implementation projects across on-premise, SaaS, and BPaaS setups, designed to support upgradable customization and customer-specific extensions using contributions.

Key Points:

- Used as the baseline for all modern ACP implementations.
- Supports on-premise, SaaS, and BPaaS models with upgradeable customization.
- Allows customer-specific customization using **contributions**.
- Serves as a **sales and demo tool** to showcase ACP capabilities quickly.
- Acts as a **clean, tested, and documented sample implementation**.
- Represents best practices for ACP customization.
- Provides a common baseline for future projects.

Components of ABR/S:

- Code
- Reference Data
- Demo Data
- Documentation
- Support

(ABR/S stands for Avaloq Banking Reference/Starter Kit, also known as Avaloq Banking Standards/Starter Kit.)