PharmaSUG China 2024 - Paper 10013

Technical Stack for Automation and Intelligence Development in Statistical Programming at BeiGene

Jinling.Li, BeiGene

ABSTRACT

In the pursuit of higher efficiency and quality, the automation and intelligent development of statistical programming has consistently been a focal point, as evidenced by numerous presentations at various conferences. These presentations offer valuable insights, aiding our understanding of the technical solutions for developing automated and intelligent products.

With the evolution of various programming languages and technological frameworks, a common challenge in building automated information management platforms is the selection of technology: how do we choose the most suitable technological solution for product development?

In this paper, I will draw upon actual development projects at BeiGene, as well as industry practices I am familiar with, to summarize the different system frameworks (C/S, B/S) currently used in automation and intelligent development, along with the relevant programming technologies. I will also discuss their application scenarios, hoping to provide readers with meaningful insights.

INTRODUCTION

The explosive growth of AI has ignited a wave of automation innovation in the pharmaceutical industry. From protocol to submission package, every stage is being explored for digitalization, automation, and intelligence. BeiGene has conducted extensive research in these areas and has introduced many industry-leading solutions for Scientific Programming automation. One such solution is the Integrated SAS Programming Platform (ISPP), a centralized platform that handles multiple functions and tasks for statistical programming.

ISPP is built on the foundations of standardization and digitalization, with the objectives of achieving visualization and intelligence. It has engaged in extensive exploration and practical implementation in the realm of automated system development. This paper will provide a comprehensive summary of the development journey of ISPP, focusing on aspects such as system architecture, technology stack selection, access control management, collaborative team development, and automated deployment processes.

SUMMARY OF SYSTEM ARCHITECTURE

There are three common types of system architectures:

Standalone Desktop Application: A standalone desktop application runs on a local computer and does not rely on a network connection. All data and logic are processed locally. This is a traditional approach and is still used for certain types of applications where network independence is crucial. ISPP 1.0 is built on this architecture.

Client/Server (C/S) Architecture: The C/S architecture is based on a client and server model, where users access the application on the server through dedicated client software. It is a well-established model used in many enterprise applications. The client software handles the user interface and some processing, while the server handles data storage and business logic. ISPP 2.x App Edition follows this architecture.

Browser/Server (B/S) Architecture: The B/S architecture is based on a browser and server model, where users access the application on the server through a web browser. It is a modern and widely adopted approach, especially for web applications. It offers the advantage of being platform-independent and easier to deploy and update. Building on ISPP 2.x App Edition, we also introduced ISPP 2.x Web Edition using this architecture.

Understanding the differences between B/S, C/S, and standalone desktop application architectures is crucial for selecting the appropriate architecture to meet various project requirements, thereby enhancing development efficiency and system performance.

The following sections will introduce the basic concepts, advantages and disadvantages, application scenarios, and technology stack choices for B/S, C/S, and standalone desktop application architectures. This will help readers gain a comprehensive understanding of these three architectures. Additionally, I will highlight the technology stack used in ISPP in blue for reference.

Dimension	Standalone Desktop Application	Client/Server (C/S) Architecture	Browser/Server (B/S) Architecture
Architecture Components	• User Interface: Provides the interface for user interaction • Business Logic: Handles the application's business logic • Data Storage: Local database or file system for data storage	• Client: User interface and part of the business logic, responsible for user interaction and initial data processing • Server: Main business logic and data storage, responsible for handling complex business logic and data management	• Browser: User interface layer, responsible for presenting data and interacting with the user • Server: Application layer and data layer, responsible for handling business logic and data storage
Advantages	• Performance: Does not rely on network connection, fast response time • Security: Data is stored locally, higher security • Independence: Can be used offline without a network connection	• Performance: Generally more efficient and faster response time than B/S architecture • Rich Functionality: Can implement more complex functions and provide a better user experience, such as direct access to the user's local file system, offering more user-friendly shortcuts • Cross-Platform: Thanks to the separation of front-end and back-end, C/S architecture can also achieve good cross-platform compatibility. Modern web technologies can be used for desktop development, allowing a single codebase to serve both C/S and B/S architectures	• Cross-Platform: Accessible via any browser without the need for dedicated client installation • Ease of Maintenance: Updates and maintenance are centralized on the server side, reducing client maintenance costs • Scalability: Easy to scale and upgrade, suitable for large-scale user access
Disadvantages	• Complex Maintenance: Requires updates and maintenance on each user's computer	• Complex Maintenance: Requires updates and maintenance on each client, increasing maintenance costs. Although automatic update push	• Performance: May be limited by network speed and browser performance, resulting in slower response times

	• Data Synchronization: Difficult to synchronize data across multiple devices • Computational Limitations: Local machines, typically PCs, may struggle with large-scale data and model computations • Data Isolation: In the AI era, data is crucial. Traditional standalone applications create data silos, significantly reducing the value of data, and are largely obsolete	features can be added, they may impact user experience • Cross-Platform Limitations: While modern front-end technologies can serve different platform clients with a single codebase, perfect adaptation to each platform and terminal requires code adjustments and optimizations, increasing development costs	• Security: Data transmission may pose security risks, requiring additional security measures
Typical Applications	• Notepad++ • ISPP Version 1.X	• Microsoft Teams • ISPP Version 2.X App Edition	• EDC Systems (Rave) • ISPP Version 2.X Web Edition
Technology Stack	• Programming Languages: C++, C#, Java, Python, JavaScript, etc. • Frameworks: .NET, Qt, JavaFX, Tkinter, etc. • Databases: SQLite, LocalDB, file system	• Client: JavaFX, Electron, WPF, Qt, Swift (iOS), Kotlin (Android) • Server: Java (Spring), .NET, Node.js (Express), Python (Django/Flask/FastAPI) • Databases: Oracle, SQL Server, MySQL, PostgreSQL	• Front-End: HTML, CSS, JavaScript, React, Angular, Vue.js • Back-End: Node.js, Django, Ruby on Rails, ASP.NET, Spring Boot • Databases: MySQL, PostgreSQL, MongoDB, Redis • Integrated Frameworks: Streamlit, Shiny, which allow data scientists and product managers to quickly build prototypes or POCs. Although they can embed front-end technologies like CSS, JavaScript, and HTML, they are limited compared to modern web development frameworks

Performance	• Response Time: Standalone desktop applications typically have the fastest response time as all processing is done locally, though they cannot handle large-scale data and model computations • Resource Consumption: Standalone desktop applications and C/S architecture typically consume more local resources as part of the business logic is processed locally	• Response Time: C/S architecture is generally faster than B/S but still limited by network speed and server performance • Resource Consumption: Standalone desktop applications and C/S architecture typically consume more local resources as part of the business logic is processed locally	• Response Time: B/S architecture is the slowest • Resource Consumption: B/S architecture consumes fewer local resources as most business logic is processed on the server side
Security	• Data Transmission Security: Highest in standalone desktop applications as data is stored locally• System Vulnerabilities: Prone to local vulnerabilities	• Data Transmission Security: High in C/S architecture • System Vulnerabilities: Prone to client-side vulnerabilities	• Data Transmission Security: Lower in B/S architecture, requiring secure protocols like HTTPS • System Vulnerabilities: Prone to attacks like XSS and CSRF
Maintenance and Scalability	• Maintenance Costs: High for standalone desktop applications as updates and maintenance are required on each client • Scalability: Poor scalability, suitable for small-scale user access	• Maintenance Costs: High for C/S architecture as updates and maintenance are required on each client • Scalability: Poor scalability, suitable for small-scale user access	• Maintenance Costs: Low for B/S architecture as updates and maintenance are centralized on the server side • Scalability: Good scalability, suitable for large-scale user access

CHOOSING THE APPROPRIATE ARCHITECTURE

Requirement Analysis

User Requirements: When selecting an appropriate architecture, it is crucial to consider user needs. For applications that require cross-platform access, a Browser/Server (B/S) architecture is ideal. This architecture allows users to access the application from various devices and operating systems through a web browser. On the other hand, applications that demand high performance and complex functionalities are better suited for a Client/Server (C/S) architecture. This setup can leverage the processing power of client machines to handle intensive tasks. For applications that need to function offline, a standalone desktop application is the best choice, as it does not rely on continuous internet connectivity.

System Requirements: From a system perspective, scalability and security are key considerations. Systems that require high scalability should opt for a B/S architecture, as it can easily accommodate an increasing number of users and data. Conversely, systems that prioritize high security should consider a C/S architecture, which can offer more robust security measures by controlling data flow between the client and server. For systems that need to operate independently without network dependencies, a standalone desktop application is the most suitable option.

Data Requirements: When it comes to data, the choice of architecture depends on whether the system needs to collect and share business data. Systems that require data sharing and collaboration are best served by either B/S or C/S architectures, as they facilitate data synchronization and accessibility. If data sharing is not a priority, a standalone desktop application can suffice, providing a simpler and more isolated environment.

Project Scale

The scale of the project also influences the choice of architecture. Small-scale projects often benefit from B/S architecture or standalone desktop applications due to their lower development and maintenance costs. Large-scale projects, however, may require a C/S architecture to achieve higher performance and support more complex functionalities.

Technical Team

The expertise of the technical team plays a significant role in architecture selection. If the team excels in web development, a B/S architecture is a natural fit. For teams proficient in desktop or mobile development, a C/S architecture is more appropriate. Teams with strong local development skills may find standalone desktop applications to be the best match for their capabilities.

Development Timeline

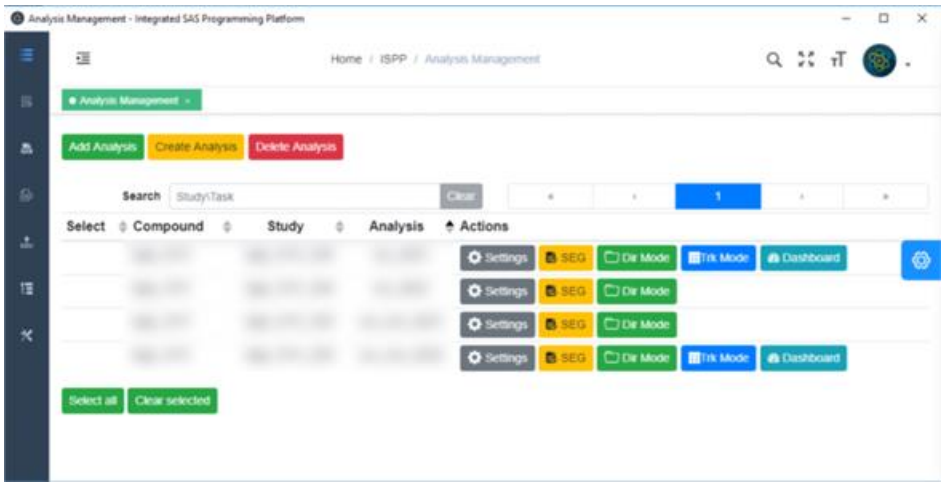
The development timeline is another critical factor. Projects with shorter development cycles are well-suited for B/S architecture, as web applications can often be developed and deployed more quickly. Projects with longer development timelines can consider C/S architecture or standalone desktop applications, which may require more time to develop but can offer greater performance and functionality.

In conclusion, selecting the right technical stack and architecture involves a careful analysis of user needs, system requirements, data considerations, project scale, team expertise, and development timelines. By aligning these factors with the strengths of B/S, C/S, and standalone desktop architectures, you can ensure that your system is both effective and efficient.

CASE STUDY

Standalone Desktop Application Case: ISPP 1.X

ISPP 1.X is a standalone desktop application designed to simplify switching between analysis folders and provide programmers with a user-friendly interface for batch running programs. Additionally, ISPP offers a dashboard to monitor the status of batch runs and display alerts if any issues are detected in log or lst files, or if relevant files are not in the planned chronological order.

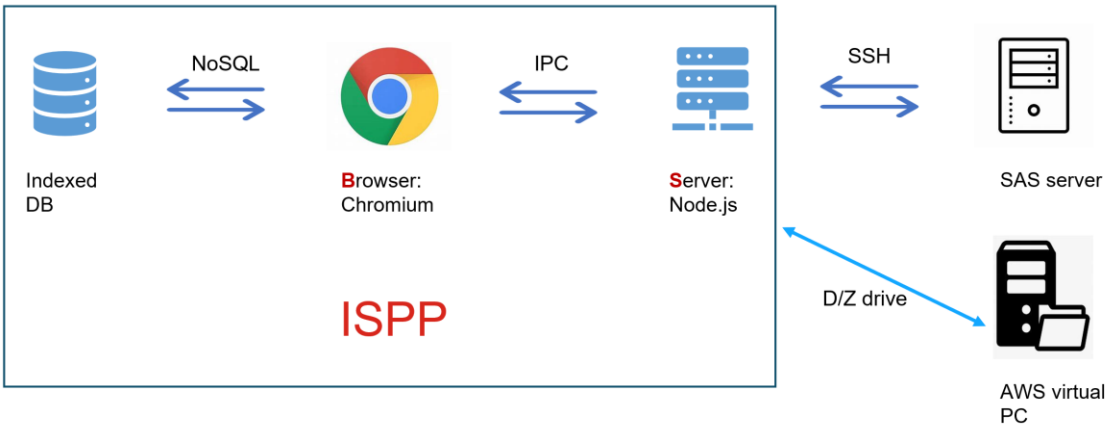


ISPP 1.X is developed entirely in JavaScript, utilizing the following architecture. IndexedDB, a NoSQL database, provides data storage services on the frontend.

Within the Electron.js framework, there are two main components: Chromium and Node.js. Chromium serves as the user interface for ISPP, while Node.js functions as the server-side component. These two components communicate via Inter-Process Communication (IPC).

User data is stored in IndexedDB, which interacts with Chromium. Node.js can communicate with the SAS server via SSH and can also interact with the user's local file system, whether it is an AWS virtual machine or a physical laptop.

The general workflow of ISPP is as follows: retrieve program information from the user's local PC, submit the program information to the SAS server, perform checks, store the information in IndexedDB, and present the information to the user through Chromium.



ISPP does not require any commercial server or database. Instead, it must be installed on the user's local laptop. User settings and operational data are stored on the user's working laptop, which presents several challenges:

- Users lose all data and settings if their Amazon Workspace is rebuilt.
- Data cannot be shared between users via ISPP.
- Implementation of authorization and authentication is not possible.

Client/Server (C/S) Architecture Case: ISPP 2.X App Edition

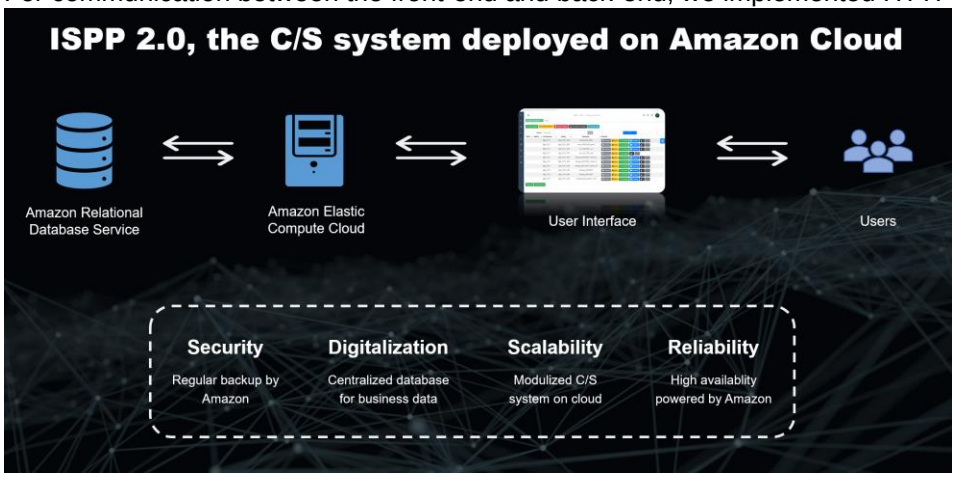
To address the challenges encountered with the standalone version of ISPP, we initiated a major overhaul in 2022 to upgrade it to a Client/Server (C/S) architecture. The new architecture adopts a front-end and back-end separation model, utilizing the classic MVC pattern, where M stands for Model, V for View, and C for Controller. The entire back-end, which includes both the Model and Controller, is deployed on Amazon Cloud.

For the View, we adopted the Single Page Application (SPA) development model, using Vue.js as our web development framework. Specifically, we employed Vue version 2.6, including Vuex and Vue Router, and used Webpack for bundling.

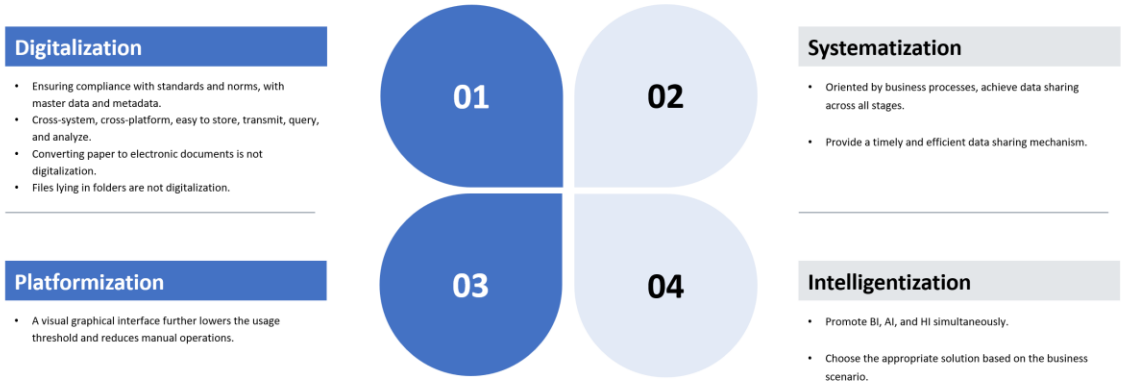
During development, we leveraged several component libraries such as Element UI, Bootstrap, and ECharts. These libraries significantly enhance front-end development efficiency, allowing us to quickly build user-friendly UIs.

Finally, we used Electron.js to package the SPA as a desktop application, which is then distributed to users for installation and use.

For communication between the front-end and back-end, we implemented HTTPS.

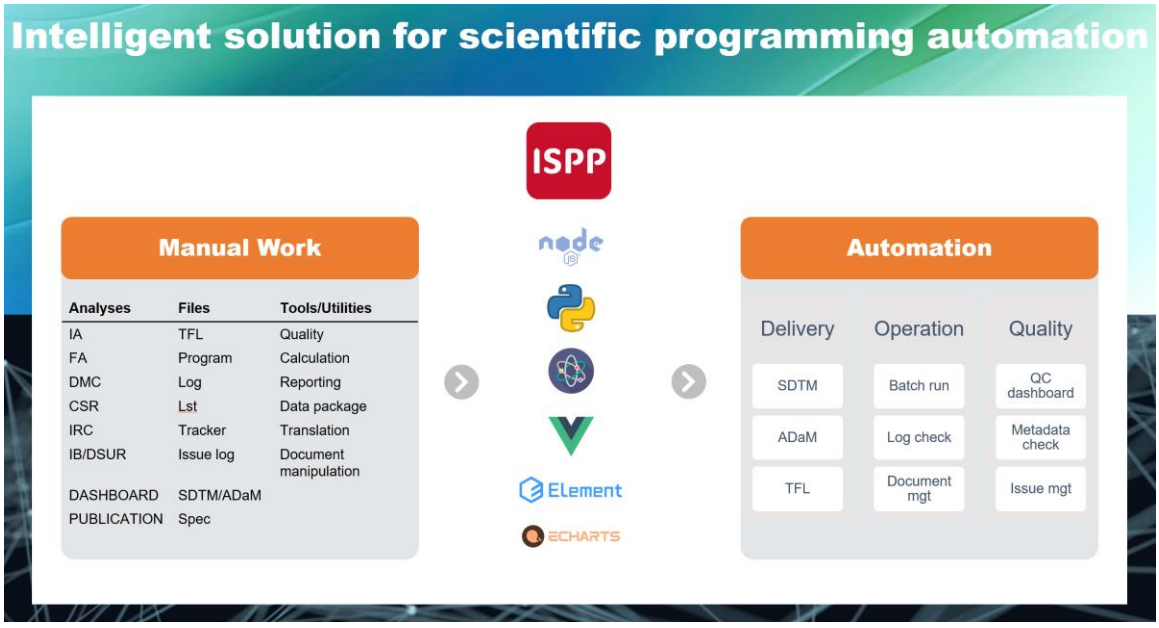


Our goal is to create a one-stop ecosystem for clinical data statistical analysis by implementing digitalization, systematization, platformization, and intelligence. To achieve this, we have expanded the scope of ISPP's business operations.



In addition to the existing features such as batch run, log checking, and folder management, we have integrated automation modules for the entire workflow—from SDTM to ADaM, TFL, and the final submission package. We have also incorporated a comprehensive quality control module throughout the process.

Some of these modules are already live and serving our clinical trials, while others are still under development.

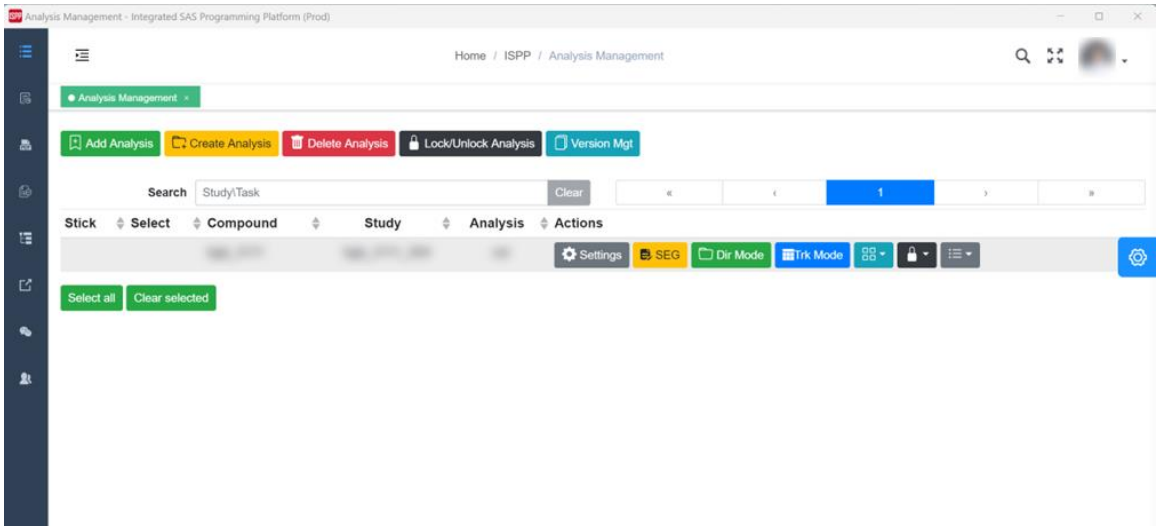


As mentioned earlier, ISPP is a Client/Server (C/S) ecosystem. On the server side, our technology stack includes Python and the Flask web development framework. Within Flask, we utilize a powerful database middleware called Object-Relational Mapping (ORM). ORM is a programming technique used to map objects in object-oriented programming languages to data in relational databases. Its primary purpose is to simplify database operations, allowing developers to interact with the database using an object-oriented approach without writing extensive SQL queries.

For our databases, we primarily use serverless SQLite and Microsoft SQL Server. The SQL Server is procured through Amazon Cloud's Relational Database Service (RDS).

On the front end, our main language is JavaScript, complemented by CSS and HTML. These three technologies are essential for web development and creating user interfaces.

The following image shows the home page of the upgraded ISPP 2.X C/S architecture.



During the development of the C/S architecture, we encountered several challenges:

User Account and Permission Management: We cannot store individual user passwords in the database because users might set their personal and company passwords to be the same. This poses a risk as our database administrators could potentially see these passwords. Even with secondary encryption, this issue cannot be entirely mitigated. Therefore, we need to implement Single Sign-On (SSO) using the company's account management system for user authentication. We will use Microsoft's Azure Active Directory accounts for SSO, ensuring that no user password information is stored in the database.

File System Interaction: Statistical programmers frequently need to interact with the file system, such as generating, deleting, opening, uploading various types of files, and editing or reviewing documents or programs. However, colleagues from other departments do not require extensive file operations and prefer simple and quick actions via a web interface without needing to install specific client software.

Client Installation: Each time a new version is released, users need to install the updated version, which can negatively impact the user experience.

To address these issues and provide greater convenience, we are advancing the development of a web-based ISPP using a Browser/Server (B/S) architecture.

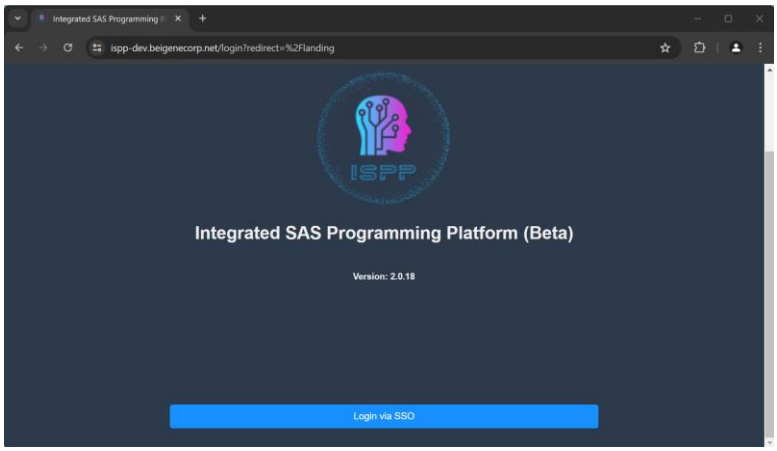
Browser/Server (B/S) Architecture Case: ISPP 2.X Web Edition

Due to browser limitations, web applications run within the browser's sandbox environment. This means that the browser cannot directly access the user's local file system (with exceptions for temporary files stored via cookies, local storage, session storage, etc.). These restrictions necessitated some functional reductions in the ISPP 2.0 app version when transitioning from a C/S to a B/S architecture.

For instance, functionalities such as batch runs and opening, editing, and reviewing files within the file system had to be reduced, as the B/S version of ISPP currently cannot connect to our SAS server. However, features that do not rely on the SAS server have been retained in the web version, especially those related to cross-departmental collaboration.

Additionally, we have introduced new automation modules to facilitate cross-departmental collaboration.

The following image shows the login interface of the ISPP web version. Notably, we have integrated with the company's system accounts to enable Single Sign-On (SSO).



The web version based on the B/S architecture shares almost the same backend code as the C/S architecture app version. Thanks to the separation of frontend and backend development, the backend can serve both the B/S architecture web version and the C/S architecture app version seamlessly.

For the frontend, the code is also largely shared between the two architectures. As previously mentioned, we use Electron to package a Single Page Application (SPA) developed with Vue.js for the client. By removing Electron and directly deploying the SPA, we achieve the B/S architecture web version.

This approach of using a single codebase to serve both architectures significantly accelerates development efficiency, reduces development costs, and enhances system scalability.

ACCESS CONTROL AND USER AUTHENTICATION

Whether it's a B/S or C/S architecture system, one unavoidable issue is user authentication and access control. Both architectures store data and resources on the backend, and we need to ensure that access to these resources is controlled. Only authenticated users should be able to access specific authorized resources, and their usage should be logged. Authentication and authorization are also fundamental for version control and audit trails.

Every type of business system requires this functionality. However, if each system independently develops its own user authentication and authorization module, it leads to inefficient and redundant development efforts. The current trend is to leverage third-party authentication platforms for unified user management, commonly known as Single Sign-On (SSO). ISPP currently utilizes the company's account management system deployed on Microsoft Azure Active Directory to achieve SSO.

To implement SSO, you need to register your system on Azure Active Directory. Once you obtain the necessary information and permissions, you can apply these credentials within your system to

enable SSO. Typically, you will need to contact your IT department to help register your application.

Azure Active Directory supports multiple platforms and languages, allowing you to implement SSO in almost any popular language and architecture. For instance, ISPP has already achieved SSO for both its App & and Web versions.

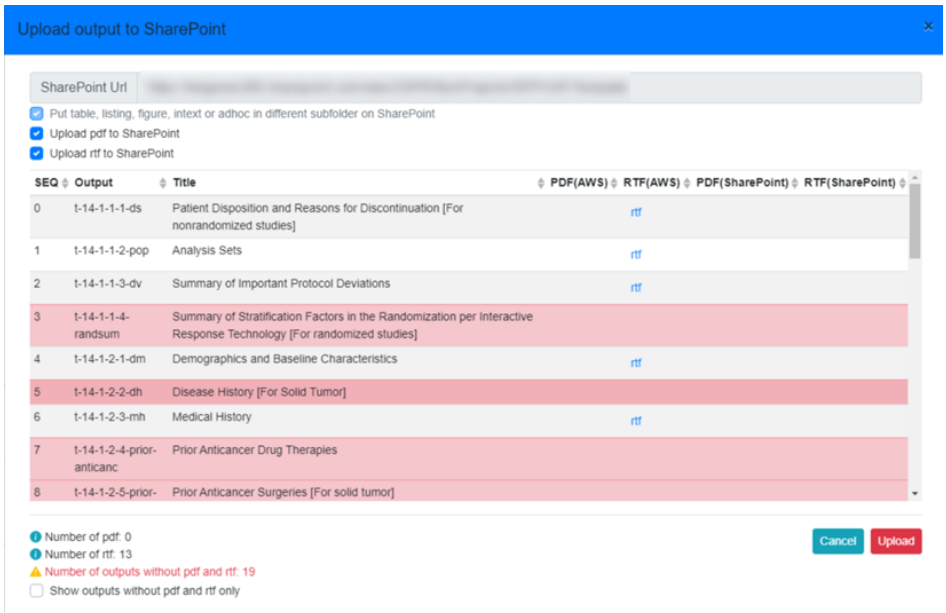
For detailed implementation methods, please refer to: Microsoft identity platform documentation - Microsoft identity platform | Microsoft Learn.

By leveraging Azure Active Directory, we ensure that our user authentication and access control are robust, scalable, and efficient, providing a seamless experience across different system architectures.

Once Single Sign-On (SSO) is implemented, we can obtain ID tokens and access tokens issued by Microsoft. These tokens can be used to access Microsoft's Graph API, enabling integration with Office applications.

For example, ISPP can now automatically upload files to SharePoint by using the access token to interact with Microsoft's SharePoint Graph API. The Microsoft Graph API is highly comprehensive, allowing ISPP to integrate numerous Office functionalities. This facilitates seamless information transfer across systems, such as sending emails, checking mailboxes, and sending and viewing Teams messages.

Below is the UI for ISPP's one-click file upload to SharePoint:



By leveraging the capabilities of the Microsoft Graph API, ISPP enhances its functionality and improves cross-system communication, making it a more powerful and versatile tool for users. This integration not only streamlines workflows but also ensures that users can efficiently manage their tasks within a unified platform.

The previous sections provided a brief overview of account management. Equally important is the issue of permission management. This is typically achieved by establishing roles and permission lists within our database. We need to predefine role categories and permission lists, considering the smallest unit of permission, the corresponding operations, or resource access rights. Once we

have organized the role and permission lists, we can implement permission management by associating users with specific roles, thereby configuring different permissions for different users.

In cases where we have a wide variety of resources, each with different management models, we can create distinct permission lists for different resource categories. By establishing an intermediary relationship table between roles and permission lists, we can effectively manage permissions for various resources.

After implementing user account and permission management, we can consider adding version control and audit trails for system operations. This allows us to track user actions on resources, such as create, read, update, and delete (CRUD) operations. Essentially, it records who performed what action and when, and it allows for the possibility of undoing or even rolling back to a specific point in time.

There are several ways to achieve this. One approach is to attach a logging system to the database. Another approach is to directly implement code at the business layer to record each user action and the state of resources before and after the action. ISPP uses the latter approach because it offers greater flexibility and gives developers more control. However, this also increases the complexity of backend development logic. Fortunately, ISPP uses the ORM SQLAlchemy for backend development, and with the help of SQLAlchemy-Continuum, we can implement version control and audit trails at the ORM layer.

By leveraging these tools, ISPP ensures robust account and permission management, along with comprehensive tracking and rollback capabilities, thereby enhancing the system's reliability and security.

COLLABORATIVE TEAM DEVELOPMENT

Developing a large-scale, multi-functional, and multi-module system necessitates effective team collaboration. Efficient collaboration is paramount, and two of the most critical components are code management and automated deployment.

Code Management

Code management is a pivotal aspect of collaborative development. In a multi-developer environment, it is essential to ensure that code versioning and merging are executed efficiently and accurately. ISPP leverages GitLab for code hosting, with separate repositories for the frontend and backend. Each repository employs a structured branching strategy:

1. **Develop Branch**: This branch is the most up-to-date and comprehensive, though not necessarily the most stable. It serves as the integration branch where all feature branches are merged.
2. **Main Branch**: This branch is thoroughly tested and ready for deployment to the production environment at any time.
3. **Automated Deployment Branches**: These branches are configured for Continuous Deployment/Continuous Integration (CD/CI). Typically, there are three branches corresponding to deployment in development, testing, and production environments.
4. **Feature Branches**: These branches are generally created from the develop branch and are used for developing new features. They are eventually merged back into the develop branch.

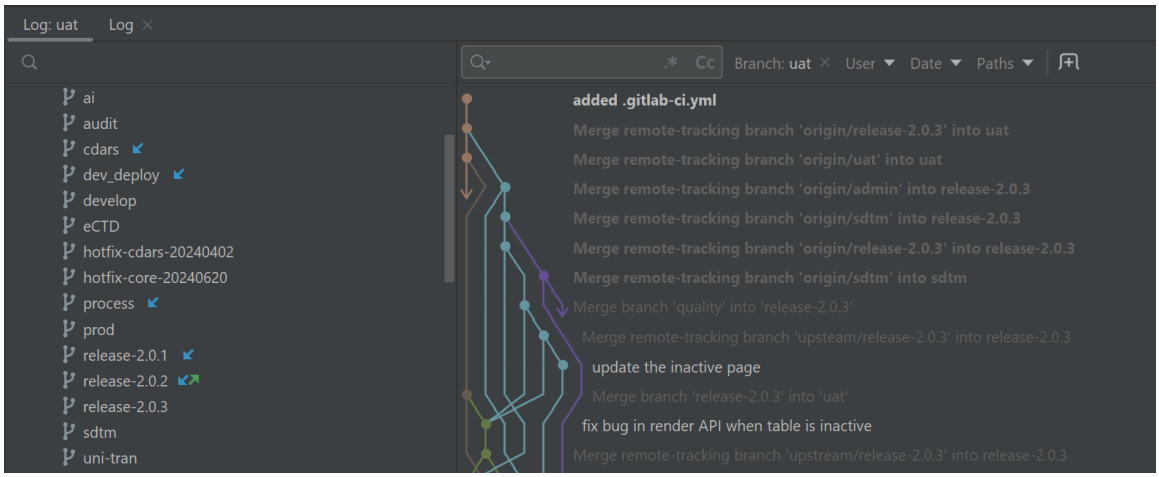
5. **Release Branches**: These branches are cut from the develop branch for new version releases. After rigorous testing, they are merged into the main branch for production deployment.

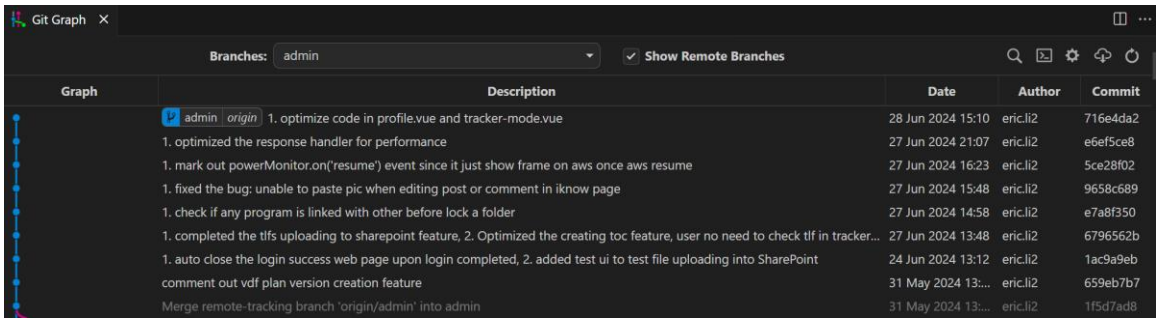
6. **Hotfix Branches**: These branches are used for urgent bug fixes in the production environment. They are usually created from the main branch and, after the fix, are merged back into both the main and develop branches.

The above outlines the fundamental principles of branch management in code management. For version control, proficiency in Git is indispensable for every software engineer. Common operations such as commit, push, pull, and fetch are frequently utilized in our daily workflows.

Many Integrated Development Environments (IDEs) integrate these operations and provide a visual interface for Git operations. However, the most stable and reliable method remains using Git commands. Regardless of the IDE, the underlying commands are consistent, but the UI can vary and may introduce complexity. This complexity can lead to errors, such as incorrect merges, which can disrupt the version chain. Such issues have indeed occurred during ISPP development.

Below are the visual Git operation interfaces provided by PyCharm and VS Code (via plugins):





Code Standards

Another critical element of collaborative development is adherence to code standards. Establishing and maintaining a set of code standards before commencing development is crucial to ensure code robustness, readability, and extensibility.

ISPP has established unified frontend and backend code management standards, encompassing basic principles and best practices. These standards cover naming conventions for variables, classes, constants, files, and folders, among other elements.

By adhering to these standards and practices, we ensure that our codebase remains clean, maintainable, and scalable, facilitating smoother collaboration and higher-quality software development.

Automated Deployment

Rapid development, quick iteration, and swift deployment are essential requirements in system development. Leveraging GitLab, we have implemented CD/CI (Continuous Deployment/Continuous Integration) pipelines. Whenever a merge or push operation is initiated on the relevant branches, the pipeline is automatically triggered, deploying the code to different environments. As mentioned earlier, we have three distinct environments: production, testing, and development. Correspondingly, we have configured pipeline repository branches for each environment.

The basic steps to configure a pipeline are as follows:

1. **Create Deployment Scripts on Your Server**: For example, ISPP's deployment script might look like this:

```
#!/bin/bash
cd /srv/ispp/ispp-back-end/
pgrep -f 'gunicorn.*8000' | xargs -r kill
pgrep -f 'gunicorn.*8001' | xargs -r kill
source ./venv/bin/activate
pip install -r requirements.txt
gunicorn -w 3 ispp:app --bind 0.0.0.0:8000 --daemon --timeout 600
gunicorn --worker-class eventlet -w 1 ispp:app --bind 0.0.0.0:8001 --daemon --timeout 600
deactivate
```

2. **Configure GitLab Pipeline Runner**: This usually requires support from the IT department to set up the runner on GitLab.
3. **Create SSH Keys on the Deployment Server**: These keys facilitate secure communication between GitLab and your server. You need to add the SSH key to GitLab.
4. **Set Up Variables in GitLab**: These variables store the server's IP address, username, and the path to the deployment script on the server.
5. **Create ".gitlab-ci.yml" in the Relevant Repository Branches**: GitLab provides documentation on creating this file. Users can refer to these guidelines to set up their configuration.

Below is an example of ISPP's .gitlab-ci.yml file:

.gitlab-ci.yml 2.04 KiB

Blame | Edit | Lock | Replace | Delete

```
# This file is a template, and might need editing before it works on your project.
# This is a sample GitLab CI/CD configuration file that should run without any modifications.
# It demonstrates a basic 3 stage CI/CD pipeline. Instead of real tests or scripts,
# it uses echo commands to simulate the pipeline execution.
#
# A pipeline is composed of independent jobs that run scripts, grouped into stages.
# Stages run in sequential order, but jobs within stages run in parallel.
#
# For more information, see: https://docs.gitlab.com/ee/ci/yaml/index.html#stages
#
# You can copy and paste this template into a new `.gitlab-ci.yml` file.
# You should not add this template to an existing `.gitlab-ci.yml` file by using the `include:` keyword.
#
# To contribute improvements to CI/CD templates, please follow the Development guide at:
# https://docs.gitlab.com/ee/development/cicd/templates.html
# This specific template is located at:
# https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Getting-Started.gitlab-ci.yml

stages:          # List of stages for jobs, and their order of execution
  - deploy


deploy-job:      # This job runs in the deploy stage.
  stage: deploy  # It only runs when *both* jobs in the test stage complete successfully.
  image: python:3.6.8-alpine
  script:
    - echo "====== Deploy to ISPP backend UAT server ======"
    - apk update && apk upgrade
    - apk add openssh bash
    - apk add rsync
    - eval $(ssh-agent -s)
    - echo "$SSH_ISPP_DEV_PRIVATE_KEY" | ssh-add -
    - mkdir -p ~/.ssh
    - '[[ -f /.dockerenv ]] && echo -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config'
    - echo "Deploying to $ISPP_DEV_VM_IPADDRESS"
    - rsync -av --progress --exclude='.git' ./ $SSH_ISPP_USER@$ISPP_DEV_VM_IPADDRESS:/srv/ispp/ispp-back-end/
    - echo "Uploading ISPP backend UAT files"
    - ssh $SSH_ISPP_USER@$ISPP_DEV_VM_IPADDRESS "/home/svc.ispp@beigenecorp.net/deployment_script.sh"
    - echo "Finished deploying ISPP UAT Backend"
  only:
  - uat
  tags:
    - docker
  retry:
    max: 2
    when: runner_unsupported

```

Additionally, the CI/CD variables added in GitLab are crucial. It is important to note that these variables can only be used in protected branches. Therefore, only protected branches can serve as automated deployment branches.

Variables

Collapse

Variables store information that you can use in job scripts. Each project can define a maximum of 8000 variables. Learn more.

Variables can be accidentally exposed in a job log, or maliciously sent to a third party server. The masked variable feature can help reduce the risk of accidentally exposing variable values, but is not a guaranteed method to prevent malicious users from accessing variables. How can I make my variables more secure?

Variables can have several attributes. Learn more.

- `Protected:` Only exposed to protected branches or protected tags.
- `Masked:` Hidden in job logs. Must match masking requirements.
- `Expanded:` Variables with `$` will be treated as the start of a reference to another variable.

CI/CD Variables </> 5 — Reveal values | Add variable

Key ↑	Value	Environments	Actions
ISPP_DEV_VM_IPADDRESS Protected Expanded	*****	All (default)	
ISPP_PROD_VM_IPADDRESS Protected Expanded	*****	All (default)	
SSH_ISPP_DEV_PRIVATE_KEY Protected Expanded	*****	All (default)	
SSH_ISPP_PROD_PRIVATE_KEY Protected Expanded	*****	All (default)	
SSH_ISPP_USER Protected Expanded	*****	All (default)	

By following these steps, ISPP ensures a streamlined and efficient automated deployment process, facilitating rapid development cycles and minimizing downtime. This approach not only enhances productivity but also ensures that deployments are consistent and reliable across different environments.

FUTURE DEVELOPMENT TRENDS

In recent years, emerging technologies such as AI, cloud computing, microservices, and containerization have significantly impacted B/S (Browser/Server) and C/S (Client/Server) architectures. These advancements have simplified and streamlined the development and deployment processes.

Hybrid Architecture

The hybrid architecture development model allows B/S and C/S systems to share a substantial portion of their codebase. For instance, the ISPP 2.0 application and web platform mentioned earlier are examples of hybrid development. We utilized Electron to develop cross-platform desktop applications, seamlessly integrating web technologies with desktop capabilities.

Artificial Intelligence and Machine Learning

AI and machine learning are transforming how applications are developed and deployed. These technologies enable predictive analytics, natural language processing, and intelligent automation, which can significantly enhance user experience and operational efficiency. Integrating AI into B/S and C/S architectures can lead to more intelligent and responsive systems.

Cloud Computing

Cloud computing continues to revolutionize the way applications are hosted and managed. With cloud services, developers can leverage scalable infrastructure, reduce operational costs, and improve system reliability. Cloud-native applications, which are designed to fully exploit cloud environments, are becoming the norm. This trend is driving the adoption of microservices and serverless architectures.

Microservices

Microservices architecture breaks down applications into smaller, independent services that can be developed, deployed, and scaled individually. This approach enhances flexibility, allows for continuous delivery, and improves fault isolation. Microservices are particularly beneficial for large-scale applications, enabling teams to work on different services concurrently and deploy updates without affecting the entire system.

Containerization

Containerization, with tools like Docker and Kubernetes, has become a cornerstone of modern application deployment. Containers encapsulate applications and their dependencies, ensuring consistency across development, testing, and production environments. Kubernetes, in particular, provides powerful orchestration capabilities, automating deployment, scaling, and management of containerized applications.

Edge Computing

Edge computing is gaining traction as a way to process data closer to where it is generated, reducing latency and bandwidth usage. This is particularly important for applications requiring real-time processing, such as IoT devices and autonomous systems. By integrating edge computing with B/S and C/S architectures, developers can create more responsive and efficient applications.

Security Enhancements

As cyber threats become more sophisticated, enhancing security measures is paramount. Future development trends include the adoption of zero-trust security models, advanced encryption techniques, and AI-driven threat detection. Ensuring robust security in B/S and C/S architectures will be critical to protecting sensitive data and maintaining user trust.

The future of B/S and C/S architectures is being shaped by a confluence of advanced technologies. Hybrid architectures, AI, cloud computing, microservices, containerization, edge computing, and enhanced security measures are driving the evolution of these systems. By embracing these trends, you can create more efficient, scalable, and secure applications that meet the demands of modern users and businesses. The integration of these technologies will not only simplify development and deployment processes but also pave the way for innovative solutions and improved user experiences.

CONCLUSION

This paper provides a comprehensive explanation and comparison of three architectures, along with an in-depth introduction to the ISPP technology stack, its iterative process, and aspects such as authorization & authentication, system integration, collaborative development, and automated deployment. Additionally, it offers a preliminary analysis of future development trends, with the hope of inspiring readers.

It is important to note that the views expressed in this paper are solely those of the author and do not represent the views of the employing company. Due to the limitations of the author's knowledge and experience, this paper does not guarantee that the methods described will yield the same results in the readers' projects. Any errors are welcome to be pointed out and corrected by the readers.

REFERENCES

Website Microsoft identity platform documentation - Microsoft identity platform | Microsoft Learn: https://learn.microsoft.com/en-us/azure/active-directory/develop/

ACKNOWLEDGMENTS

I would like to extend my sincere gratitude to all ISPP developers. Their exceptional contributions have been instrumental in creating and perfecting the powerful ISPP product.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Jinling.Li
BeiGene, Ltd.
Jinling.Li@beigene.com